INTRODUÇÃO À ORGANIZAÇÃO E À PROGRAMAÇÃO DE COMPUTADORES

USANDO LOGISIM E SCILAB

OSVALDO CARVALHO

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

UFMG

1º SEMESTRE DE 2014

Conteúdo

1 Computadores e Informação 12
1.1 Computadores 12
1.2 Informação Analógica e Digital 18
1.3 Computadores e Informação 21
1.4 Conversões análogo-digital e digital-analógica 23
1.5 Sensores e atuadores 28
1.6 Memórias 31
1.7 Organização do Conteúdo 35
2 Circuitos Combinatórios 42
2.1 Bits e códigos 42
2.2 Álgebra de Boole 47
2.3 Portas Lógicas e Transistores 50
2.4 Introdução ao Logisim 55
2.5 Soma de duas parcelas de 1 bit 61
2.6 Soma de dois inteiros com vários bits 65
2.7 Modularização e Síntese Automática de Circuitos Combinatórios no Logisim 69
2.8 Síntese de um display hexadecimal 75
2.9 Comparação de binários sem sinal 77
2.10 Multiplexadores, Demultiplexadores e Decodificadores 81
3 Circuitos Sequenciais 84
3.1 Flip-flops e Registradores 84
3.2 Barramentos e Controle de Fluxo de Dados 88
3.3 Memórias 89
3.4 Acumuladores e Loops 91
3.5 Uma Calculadora 92
3.6 Clocks 94
4 Processadores 98
4.1 Programa Armazenado 98
4.2 CPU-0: Um Primeiro Processador 101
4.3 CPU-1: Processador com Instruções de Desvio 106
4.4 CPU-2: Processador com Instruções de Desvio Condicional 109
4.5 CPUs Reais: Instruções e Programação em Assembler 111

5 Programas Scilab 118
5.1 Compiladores, Interpretadores e Sistemas Operacionais 118
5.2 Scilab 121
5.3 Variáveis e Comandos de Atribuição 122
5.4 Usando o Scilab como Calculadora 127
5.5 Programas Scilab 132
5.6 Valores Lógicos e Strings 138
5.7 Os comandos `if-then-else` e `printf` 141
6 Matrizes 145
6.1 Matrizes e Comandos de Atribuição 145
6.2 Aritmética matricial 148
6.3 Construindo matrizes 152
6.4 Matrizes e Gráficos 154
6.5 Gráficos 3D 157
6.6 Matrizes de Strings 159
6.7 Matrizes e Expressões Lógicas 161
7 Loops 163
7.1 O comando `while` 163
7.2 Loop para o Cálculo de Fatorial 164
7.3 Loop para o Cálculo da Soma dos Elementos de um Vetor 165
7.4 Loop para encontrar o Menor Valor presente em um Vetor 166
7.5 Loop para encontrar o Máximo Divisor Comum usando o Algoritmo de Euclides 167
7.6 O comando `for` 169
7.7 Geração e Impressão de Tabelas 170
7.8 Comandos Aninhados 173
7.9 Geração e Impressão de uma Tabuada de Multiplicação 175
7.10 Arquivos 176
7.11 Matrizes de Strings e Arquivos 178
7.12 Matrizes Numéricas e Arquivos 179
7.13 Desenhando Mapas 181
8 Funções 185
8.1 Funções e Reaproveitamento de Código 185
8.2 Forma e Funcionamento 187
8.3 Funções, arquivos fonte e o Scilab 189
8.4 Funções e Encapsulamento de Detalhes (e mais sobre arquivos) 190
8.5 Funções e Desenvolvimento Top-down 193

8.6 Funções Recursivas 195
9 Algoritmos, Pesquisa 198
9.1 Definição e Características 198
9.2 Fatoração 200
9.3 Melhorando a Fatoração 204
9.4 Pesquisa 206
9.5 Pesquisa Seqüencial 206
9.6 Pesquisa Binária 207
10 Ordenação 210
10.1 Seleção e Troca 210
10.2 Intercalação (MergeSort) 215
10.3 Partição (QuickSort) 219
10.4 Dividir para Conquistar 225
10.5 Ordenação por Radical 225
11 Algoritmos Numéricos 228
11.1 Integração por Trapézios 228
11.2 Bisseção 232
11.3 Série de Taylor para exp(x) e Cancelamento Catastrófico 237
12 Complexidade de Problemas 242
12.1 Complexidade da Ordenação 242
12.2 Problemas NP-completos: O Problema do Caixeiro Viajante 246
12.3 Problemas indecidíveis: O Problema da Correspondência de Post 253
13 Próximos Passos 260
Índice Remissivo 264
Referências 267
Apêndice A: A CPU Pipoca 271
A.1 Ciclo de Instrução 272
1 Instruções 274
2 Programação em Assembler 276
3 Montagem do Programa Executável 278
4 O Circuito Principal da CPU Pipoca 279
5 O Contador de Programa 283
6 A Unidade de Controle 283
7 O Ciclo de Micro-Instrução 287
8 O Micro-Programa 292
9 A Planilha Pipoca.xls 295

Conteúdo

Prefácio

Programação de Computadores é a disciplina mais tradicionalmente oferecida por departamentos de Ciência da Computação, e geralmente também a de maior número de matrículas. Na UFMG, Programação de Computadores já era oferecida para Ciências e Engenharias alguns anos antes da criação do departamento. Seu código é DCC001, o que dá uma ideia de sua venerabilidade. Este texto é parte integrante de um conjunto de materiais pedagógicos para esta disciplina, produzidos com a intenção de serem usados em um curso de 60 horas-aula, dado em um semestre letivo. Exercícios, imprescindíveis para um bom aprendizado, não estão aqui, mas em um site Moodle que contém também apresentações, vídeos e outros materiais.

O enfoque adotado apresenta algumas características que o distinguem da forma tradicional como a disciplina é ministrada:

- O curso necessita de um professor e de monitores;
- Em cada semana são previstas uma aula expositiva, dada pelo professor, e uma aula prática, dada pelos monitores;
- Aulas expositivas podem ser dadas em auditório ou sala de aula com grande capacidade;
- Aulas práticas são oferecidas em laboratórios com computadores; melhores resultados podem ser esperados com turmas pequenas;
- Os procedimentos de avaliação do aluno incluem questionários semanais e provas, que podem (e devem) ser realizados “online”;
- As aulas práticas têm como objetivo auxiliar os alunos nas avaliações semanais;
- Uma base de questões, fechadas e abertas, está disponível no Moodle para a montagem de avaliações.
- Questões fechadas (múltipla escolha, associações, etc.) são corrigidas automaticamente; ao montar uma avaliação, o professor deve dosar o uso de questões abertas para manter o trabalho de correção em volumes aceitáveis.
- Conjuntos de exercícios podem também ser disponibilizados no Moodle, permitindo estudos e auto avaliações pelos alunos;
- A linguagem adotada é o Scilab;
- O primeiro terço do curso é dedicado ao ensino de circuitos lógicos e organização de computadores, utilizando o software Logisim [1] (Burch, 2002);
- Mesmo com um enfoque eminentemente prático, o material cobre conceitos teóricos fundamentais como complexidade de algoritmos, problemas np-completos e problemas indecidíveis.

Com relação ao conteúdo, os pontos que provavelmente necessitam de maiores justificativas são (1) o estudo de organização de computadores, (2) a adoção de Scilab, e (3) a inclusão de material sobre complexidade e indecidibilidade.

Qualquer curso introdutório de Programação de Computadores tem que falar alguma coisa sobre computadores; nós tomamos a decisão de falar um pouco mais do que o tradicional. A nosso ver o estudo de organização de computadores abre oportunidades para a introdução de diversos conceitos fundamentais para a programação de computadores. O aluno tem contato com bits, com operações lógicas, com códigos importantes como ASCII, inteiros sem sinal e ponto flutuante. Acumuladores são o primeiro contato com loops, e somadores são um

[1] Carl Burch, “Logisim - a graphical tool for designing and simulating logic circuits”, acessado 7 de março de 2011, http://ozark.hendrix.edu/~burch/logisim/.

excelente exemplo de modularização. A execução sequencial de instruções e instruções de desvios são também elementos para a formação na mente do aluno de um mecanismo de execução de programas. E, talvez mais importante, o primeiro contato com a programação se dá sem a necessidade de abstrações com relação à possibilidade de execução de operações básicas.

Para justificar a adoção do Scilab é preciso falar um pouco sobre o fenômeno MatLab. MatLab, de *Matrix Laboratory*, é uma linguagem chamada M e um ambiente de desenvolvimento e execução voltado para aplicações técnicas. Segundo Bill McKeeman (um dos pioneiros do Computer Science Department de Stanford, hoje professor de Dartmouth e desenvolvedor da MathWorks[2], empresa que vende e mantém o MatLab), a linguagem teve origem na ideia de colocar cálculos matriciais em sua sintaxe, ao invés de utilizar chamadas de sub-rotinas como se faz em Fortran[3].

De uma forma geral, MatLab não é conhecido em departamentos de Ciência da Computação, à exceção dos que trabalham em áreas como aprendizado de máquina ou visão computacional. Não é só aqui no Brasil que isto acontece. MatLab não foi projetado por especialistas em linguagens ou compiladores, não tendo aos olhos de cientistas da computação (incluindo este autor) importância conceitual como linguagem de programação. Bryant et al[4] chegam a afirmar que Matlab "*... really is a language for manipulating matrices, not a general-purpose programming language*".

Pois mesmo não sendo uma linguagem de programação de propósito geral, MatLab tem os construtos básicos de atribuição, decisão, repetição, recursividade e modularização. Isso é largamente suficiente para os propósitos deste curso, como pode ser visto nos muitos exemplos deste texto. Além disso, o sucesso prático do MatLab é sem qualquer dúvida enorme. Ao apresentar a linguagem M para o MIT em 2005, McKeeman inicia dizendo "*as it turns out, the computer science department is about the only part of MIT that does not use MatLab*". Ele continua[5]:

> *MATLAB has grown over 20 years from academic research into a generalized tool for a wide variety of applications, including vehicle crash simulation, financial prediction, genome analysis, imbedded computer control, aircraft design and so on. More than 200 MathWorks developers are working on the next release. Another 1000 people run the rest of the business, in Natick and worldwide.*
>
> *There are about a million users. Some MATLAB users do not think of what they are doing as "programming." Some users are interested in quick and easy results. Some users want to build applications that are bullet proof and reusable over decades. I know of 100000 line MATLAB programs and MATLAB programs mining petabytes of data and others running 100x parallel for days at a time. Some universities teach MATLAB for beginning programming. On every customer contact I find new surprises. MATLAB satisfies all these communities.*

---

[2] Mathworks, "MathWorks - MATLAB and Simulink for Technical Computing", acessado 7 de março de 2011, http://www.mathworks.com/.

[3] Bill McKeeman, "MATLAB 101 -- A talk for the MIT Computer Science Department", novembro de 2005, http://www.cs.dartmouth.edu/~mckeeman/references/matlab101/matlab101.html.

[4] R.E. Bryant, K. Sutner, e M.J. Stehlik, *Introductory Computer Science Education at Carnegie Mellon University: A Deans' Perspective* (Tech. Report CMU-CS-10-140, Carnegie Mellon University, Pittsburgh, 2010, http://link. cs. cmu. edu/article. php, 2010).

[5] Bill McKeeman, "MATLAB 101 -- A talk for the MIT Computer Science Department".

Prefácio

O Scilab é um sistema livre, produzido por um consórcio de organizações francesas, que, sem tentar ser um clone, segue a mesma filosofia do MatLab. A compatibilidade das duas linguagens de programação é grande mas não total. Segundo o verbete na Wikipedia, SciLab vem conquistando cada vez mais adeptos tanto na academia como na indústria. Existem livros sobre SciLab em inglês, francês e espanhol, e está disponível na Internet um texto introdutório em português produzido pelo Prof. Paulo Sérgio da Mota Pires, da UFRN [6]. Links para esses materiais podem ser encontrados no site [7] (Scilab Consortium, s.d.).

O objetivo da inclusão de conceitos mais sofisticados é dar ao aluno noções intuitivas de complexidade e indecidibilidade, tornando-o capaz de caracterizar a eficiência de um programa de forma independente da velocidade de processadores, e aprendendo a reconhecer problemas cuja solução por computadores pode demorar muito tempo, ou mesmo demorar um tempo não limitado. A abordagem é, claro, muito introdutória, mas falamos de conceitos que constituem pilares da Ciência da Computação. Não podemos perder o que frequentemente é nossa única oportunidade de um contato maior com estudantes que não terão outras disciplinas da Ciência da Computação.

Quanto às avaliações freqüentes, a sua contribuição para a experiência de aprendizado é consensual. Dentre as principais características de cursos altamente respeitados, levantadas pelo projeto Harvard Asssesssment Seminars[8], estão:

- *High demands and standards placed upon [students], but with plentiful opportunities to revise and improve their work before it receives a grade, thereby learning from their mistakes in the process.*
- *Frequent checkpoints such as quizzes, tests, brief papers, or oral exams. The key idea is that most students feel they learn best when they receive frequent evaluations, combined with the opportunity to revise their work and improve it over time.*

Entretanto, avaliações exigem correções e, com a atual pressão por produtividade medida pela relação alunos/professor, o trabalho de correção pode se tornar insustentável. É preciso equilibrar o uso de questões fechadas, corrigidas automaticamente, com o de questões abertas, imprescindíveis em um curso de programação de computadores mas que exigem correção manual. Mesmo controlando o uso de questões fechadas, o auxílio dado por uma equipe de monitores pode ser essencial para se manter avaliações semanais.

Este texto está organizado em 12 módulos, que correspondem a 12 aulas expositivas com conteúdo novo. Isso se adapta bem a um semestre na UFMG, que tem aproximadamente 15 semanas. Com uma aula expositiva por semana, as três semanas de "folga" podem ser completadas com revisões ou outras atividades.

Agradecimentos. A metodologia adotada para Programação de Computadores (uma aula expositiva e uma aula prática por semana) e a nova linguagem (Scilab) me foram sugeridas pelos Professores Regina Helena Bastos Cabral e Ivan Moura Campos. Os dois me fizeram conhecer o Matlab, me convenceram da necessidade de práticas mais frequentes, e também da viabilidade de aplicação de provas online. A Regina já aplicava estas técnicas com enorme sucesso em Cálculo Numérico.

---

[6] "sciport-3.0.pdf", acessado 7 de março de 2011, http://www.dca.ufrn.br/~pmotta/sciport-3.0.pdf.

[7] Scilab Consortium, "Home - Scilab WebSite", acessado 7 de março de 2011, http://www.scilab.org/.

[8] Jack Chizmar, "The Effective Teaching and Learning Network - training course and information for teachers - TQM and Classroom Research", acessado 7 de março de 2011, http://www.etln.org.uk/resources/page16.html.

Os monitores que trabalharam na disciplina em sua primeira oferta em 2007 foram Maxstaley Neves e Henrique Chevreux. Sem eles eu não teria conseguido enfrentar a miríade de detalhes técnicos do Logisim, Moodle e Scilab, todos então sistemas novos para mim. Mas, muito mais do que isso, eles forjaram para a monitoria uma postura íntegra, competente e criativa, postura que souberam transmitir aos outros monitores.

A aplicação da metodologia faz uso intenso da infra-estrutura de tecnologia da informação da UFMG, competentemente administrada pelo Centro de Computação, pelo Laboratório de Computação Científica, pelo Laboratório de Recursos Computacionais do Instituto de Ciências Exatas e pelo Centro de Recursos Computacionais do Departamento de Ciência da Computação.

A figura da capa foi retirada do fac-símile do manuscrito *Dixit algorizmi* [9], tradução do árabe para latim do trabalho de al-Khwārizmī.

[9] "Muḥammad ibn Mūsā al-Khwārizmī - Wikipedia, the free encyclopedia", acessado 21 de agosto de 2012, http://en.wikipedia.org/wiki/Mu%E1%B8%A5ammad_ibn_M%C5%ABs%C4%81_al-Khw%C4%81rizm%C4%AB.

1 Computadores e Informação

1.1 Computadores

Computadores eletrônicos programáveis surgiram na década de 1940. Pouco mais de meio século após a construção dos primeiros computadores, o impacto desta tecnologia sobre nossas vidas é enorme. Computadores mudaram – e muito – a forma como se faz música, cinema ou artes plásticas, como se escrevem textos, como se ensina e se aprende, como se faz comércio, medicina, engenharia, ciência, economia, como se namora, como governar e como exercer a cidadania. Mas o que é um computador?

Vamos começar com alguns exemplos. É certo que você já viu diversos computadores, e é provável que a imagem que lhe venha à mente ainda seja similar à mostrada na Figura 1.

Figura 1: Um IBM PC, lançado em 1981[10]

Este foi o primeiro computador pessoal lançado pela IBM. Apesar de ter sido precedido desde os primeiros anos 70 por diversos outros micro-computadores, o lançamento em 1981 do IBM PC foi um marco – a IBM era então a empresa dominante na indústria da computação, provavelmente a única capaz de lançar padrões.

A evolução dos computadores foi enorme e se deu em direções de difícil previsão. O sistema operacional do IBM PC que se afirmou sobre seus concorrentes foi o MS-DOS, fornecido pela Microsoft, então uma pequena empresa que produzia compiladores para microcomputadores. O MS-DOS foi na verdade desenvolvido por Tim Patterson, da Seattle Computer Products, que por 50.000 dólares vendeu os direitos do sistema operacional que transformou a Microsoft no gigante atual. Faltou visão ao Tim Patterson? Sim, sem dúvida. Um erro entretanto mais perdoável do que o da IBM, que na época era maior núcleo de competência mundial em computação. A IBM permitiu que o contrato com a Microsoft fosse feito sem exclusividade, liberando a venda do MS-DOS aos fabricantes de clones do IBM-PC. Poucos anos depois a Microsoft se tornou uma empresa maior do que a IBM.

[10] "IBM Archives: IBM Personal Computer", acessado 7 de março de 2011, http://www-03.ibm.com/ibm/history/exhibits/pc/pc_1.html.

Existem computadores que não se assemelham a PCs. Aliás, os computadores existentes em maior número são simples e pequenos, e ficam escondidos em telefones celulares, iPods, eletrodomésticos e automóveis.

Figura 2: A maioria dos computadores existentes está embarcada em carros, eletrodomésticos, telefones, ...

Outros são enormes, ocupando vastos salões, como o Blue Gene mostrado na Figura 3. Em 2006 o Blue Gene era o computador mais poderoso do mundo, sendo capaz de executar 478 trilhões de operações aritméticas por segundo. Em 2008 já não era: o RoadRunner, que como o Blue Gene foi produzido pela IBM, atingiu 1 *petaflops*, isto é, mil trilhões, ou 10^{15} operações aritméticas por segundo. A título de comparação, em 2012, a potência de um PC de topo é da ordem de gigaflops, ou 10^{9} operações aritméticas por segundo. Como curiosidade, o RoadRunner utiliza chips (circuitos integrados) usados na PlayStation 3 da Sony.

Figura 3: O supercomputador Blue Gene [11]

[11] Lawrence Livermore National Laboratory, "BlueGene/L Photo Gallery", acessado 11 de março de 2011, https://asc.llnl.gov/computing_resources/bluegenel/photogallery.html.

A cada seis meses uma lista com os maiores computadores do mundo é publicada no site da organização Top500 [12]. Na lista de novembro de 2010, pela primeira vez o topo da lista foi ocupado por um computador chinês, o Tianhe-1A, com um desempenho de 2,6 petaflops. Na lista de novembro de 2012 o primeiro lugar é do computador Titan Cray XK7, que atingiu 17 petaflops; em novembro de 2013 a China retomou o primeiro lugar, com o Tiahne-2 que atinge 33 petaflops. Vale a pena visitar este site, que contém dados e análises interessantes sobre a evolução dos supercomputadores, denominação que a indústria dá a computadores como o Blue Gene, Tianhe-1A ou Titan.

Com a rápida evolução da eletrônica, super hoje, normal amanhã, fraco depois de amanhã. Para ter alguma utilidade o termo *supercomputador* é forçosamente relativo à época: um supercomputador é um computador que figura dentre os mais poderosos do mundo ... em sua geração.

Figura 4: O supercomputador Eniac, construído em 1946[13]

[12] "Home | TOP500 Supercomputing Sites", acessado 7 de março de 2011, http://www.top500.org/.

[13] "Computer History Museum - Timeline of Computer History", acessado 16 de fevereiro de 2012, http://www.computerhistory.org/timeline/?year=1946.

Neste sentido o Eniac, construído em 1946, e que conseguia fazer 385 multiplicações em um segundo[14], era mais super que o Blue Gene ou que o K-Computer, pois era mil vezes mais potente que seus predecessores. À guisa de comparação, o K-Computer foi apenas 4 vezes mais rápido que seus predecessores, fator que hoje é considerado muito alto ... O Eniac foi ainda mais do que mais veloz que seus predecessores: foi o primeiro computador com programa armazenado, capaz de resolver problemas genéricos.

Figura 5: Alan Turing [15]

Computadores transformam informação de entrada em informação de saída. Um fato muito importante sobre computadores é que, desde que meios de armazenamento externo de informação (discos e fitas magnéticas, por exemplo) estejam disponíveis em quantidade suficiente, todos, pequenos e grandes, são capazes de realizar *as mesmas* transformações de informação. As *computações* – nome técnico para transformações de informação – que micro e supercomputadores conseguem realizar são as mesmas que um dispositivo teórico, a *máquina de Turing*, é capaz de fazer. Este dispositivo teórico foi construído com a ambição de capturar a noção de *computabilidade*, isto é, da possibilidade de se resolver um problema de transformação de informação de forma efetiva, como uma composição de passos garantidamente realizáveis. Isso foi proposto pelo matemático inglês Alan Turing em 1937 [16], alguns anos antes do funcionamento do primeiro computador eletrônico.

As possibilidades de transformações de informação são as mesmas para computadores pequenos e grandes, rápidos e lentos, mas a velocidade com que a informação é transformada pode diferir por várias ordens de grandeza. Em muitos casos, é a velocidade que determina em o valor da informação obtida. Qualquer computador pessoal, com disponibilidade de

[14] "ENIAC - Wikipedia, the free encyclopedia", acessado 16 de fevereiro de 2012, http://en.wikipedia.org/wiki/ENIAC.

[15] "Alan Turing - Wikipedia, the free encyclopedia", acessado 11 de março de 2011, http://en.wikipedia.org/wiki/Alan_Turing.

[16] Alan M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem", *Proceedings of the London Mathematical Society* s2–42, nº 1 (1937): 230–265.

armazenamento externo suficiente, é capaz de rodar um programa de previsão meteorológica para saber como estará o tempo no dia seguinte, mas é possível que ele gaste mais do que um dia nessa tarefa, o que anularia completamente o valor da informação produzida. Da mesma forma, se um computador deve produzir informação para ser exibida como um filme, uma velocidade abaixo da necessária torna inaceitável o seu emprego.

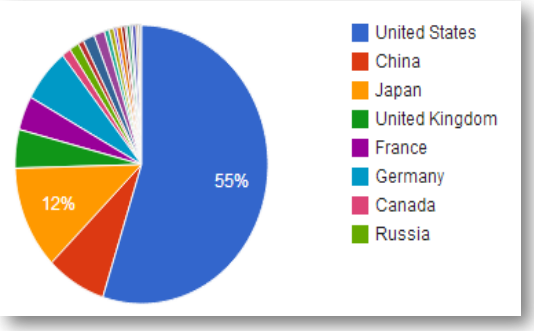


Figura 6: Participação de países em potência computacional na lista dos 500 maiores (extraído de Top500[17])

A relação entre velocidade de produção de informação e seu valor se manifesta com bastante clareza na lista dos países com maior potência computacional instalada, mostrada na Figura 6.

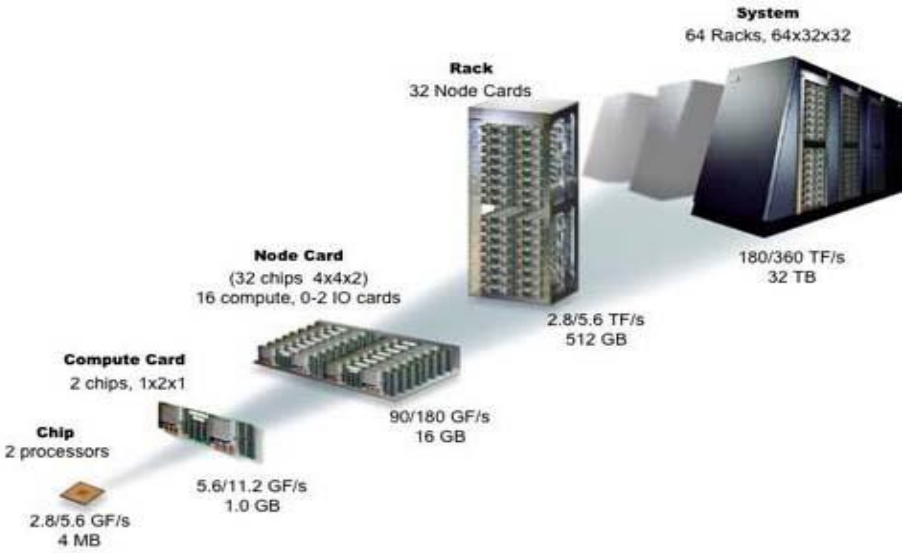


Figura 7: O Blue Gene e seus módulos[18], sub-módulos, sub-sub-módulos, ...

A Figura 7 nos permite entender melhor a estrutura do Blue Gene:

- O sistema completo tem 64 *racks*, pequenas torres que ficam abrigadas nos grandes blocos do computador.
- Cada rack tem 32 “nós computacionais”.
- Um nó computacional abriga 16 placas computacionais e, em alguns casos, até 2 placas de entrada e saída de dados.
- Cada placa computacional tem duas pastilhas (*chips*) de circuitos integrados.
- Cada chip abriga dois *processadores*, que são os circuitos que efetivamente realizam cálculos computacionais.

[17] “Home | TOP500 Supercomputing Sites”.
[18] “SciDAC Review - SIMULATION SCALE: In HPC Simulations, How Much is ENOUGH?”, acessado 2 de agosto de 2011, http://www.scidacreview.org/0801/html/scale.html.

Na Figura 7 as sucessivas decomposições param por aqui mas, na verdade, um processador é ainda uma estrutura bastante complexa.

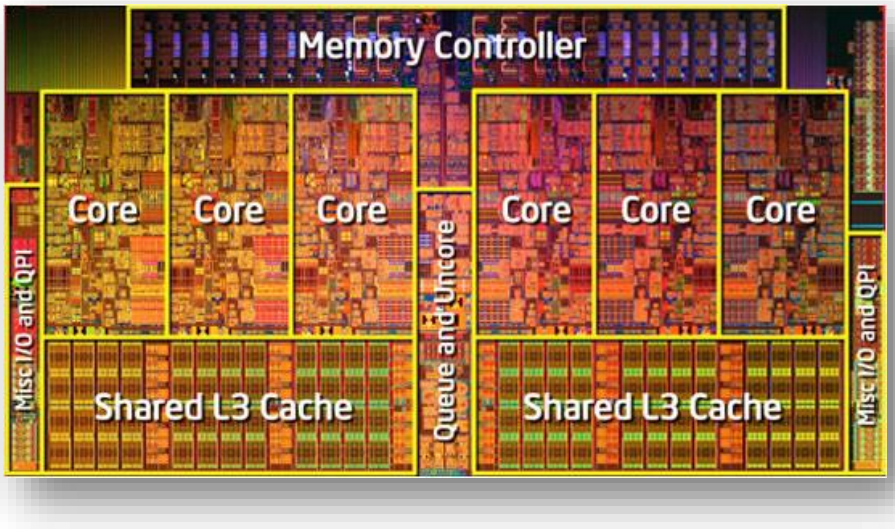


Figura 8: Chip do processador Intel Core i7-980X[19]

Olhando para a imagem (Figura 8) de um chip do processador Intel Core i7-980X (este não é o chip usado no Blue Gene), lançado em 2010, nós podemos visualmente distinguir ali diversos sub-módulos. Se pudéssemos olhar ainda mais de perto, veríamos que este chip contém nada menos que 1,17 bilhões de transistores, espremidos em uma área de apenas 248 milímetros quadrados.

Fazendo agora um zoom na direção contrária, temos a Internet, a rede mundial que conecta quase todos os computadores do mundo. A imagem da Figura 9 propositalmente lembra uma galáxia. São centenas de milhões de computadores na Internet, todos interligados.

[19] Carlos Sosa e Knudson, Brant, *IBM System Blue Gene Solution: Blue Gene/P Application Development*, IBM Redbooks SG24-7287-03, acessado 14 de fevereiro de 2012, http://www.redbooks.ibm.com/redbooks/pdfs/sg247287.pdf.

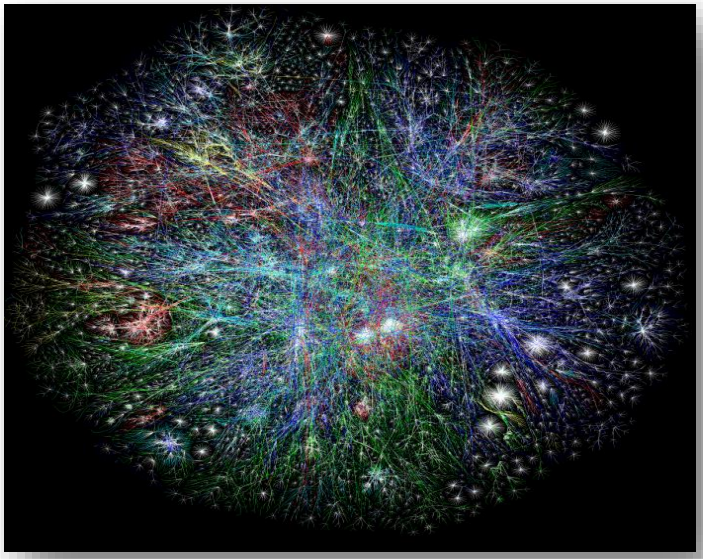


Figura 9: Uma visualização da Internet pelo produzida pelo Projeto Opte [20]. Em azul estão os nós da América do Norte, em verde os da Europa, Oriente Médio, Ásia Central e África, e em amarelo os da América Latina.

Todo esse imenso e complexo arranjo envolvendo satélites de telecomunicações, cabos submarinos, redes com e sem fios, fibras óticas, computadores, telefones celulares, circuitos integrados, incrivelmente, tudo isso funciona, com algumas falhas, é verdade, mas que não impediram a computação de mudar a forma de vida da humanidade. Funciona como? Essa proeza de organização se deve exatamente ao uso intenso de uma idéia central para a computação, a *modularização*. São muitas e muitas peças que compõem os computadores do mundo e suas redes, mas que são desenvolvidas de forma a permitir que, para utilizar um módulo, seja preciso saber apenas *o que* esse módulo faz, e não *como* ele realiza a sua função.

Módulos permitem a divisão de competências. Um programador profissional normalmente não tem a menor idéia sobre a física de semicondutores dos chips, e uma vaga idéia de como funcionam internamente os protocolos da Internet. Toda a computação é um imenso lego, onde cada peça tem encaixes bem definidos que permitem seu acoplamento para a montagem de estruturas maiores. Veremos ao longo desse curso, que módulos são onipresentes na ciência da computação, pois são fundamentais para o domínio de sistemas complexos.

1.2 Informação Analógica e Digital

Computadores trabalham com informação, e é por isso que a palavra *informática* é sinônimo de ciência da computação. Informação é um conceito de difícil definição; é algo em um objeto – o *objeto suporte* – que diz alguma coisa sobre outro objeto – o *objeto alvo* – que pode ser uma grandeza física, ou um evento localizado no tempo, ou qualquer aspecto de interesse no objeto alvo.

Quando a maneira de registro da informação no meio suporte tem uma relação física direta com o alvo da informação, como a deformação produzida em uma mola por um peso, nós dizemos que a informação ali armazenada é *informação analógica*. Quando o objeto suporte armazena *símbolos* como números ou palavras com informação sobre o objeto alvo, nós dizemos que a informação é *simbólica* ou, mais comumente, *informação digital*.

[20] Barret Lyon, “The Opte Project”, acessado 7 de março de 2011, http://opte.org/.

Figura 10: Negativos fotográficos [21]

Um filme fotográfico revelado – objeto em extinção nos dias de hoje – registra informação sobre a cena fotografada. Uma cena, fotografada com uma máquina tradicional, gera informação analógica sobre o material foto-sensível que cobre o filme. A mesma cena, fotografada com uma máquina digital, gera números que registram a intensidade de luz para cada um de muitos pontos pequeninos que compõem a imagem, números que são símbolos, ou seja, são informação digital.

Figura 11: Informação analógica e digital

Um termômetro caseiro é um objeto que fornece informação sobre a temperatura do ar, ou do corpo de uma pessoa, pois fenômenos de dilatação fazem com que a altura da coluna de mercúrio cresça com a temperatura medida. Nós dizemos que a informação obtida por um exame direto do termômetro é uma informação analógica.

[21] "Photographic film image", acessado 7 de março de 2011, http://farm4.static.flickr.com/3444/3348244651_fef16ef641.jpg.

Figura 12: Um termômetro

Com o uso de uma escala, um médico pode anotar em um papel a temperatura de seu paciente. Ao fazê-lo, a informação sobre a temperatura passa de analógica (a altura da coluna de mercúrio) para informação digital ou simbólica (os números anotados pelo médico).

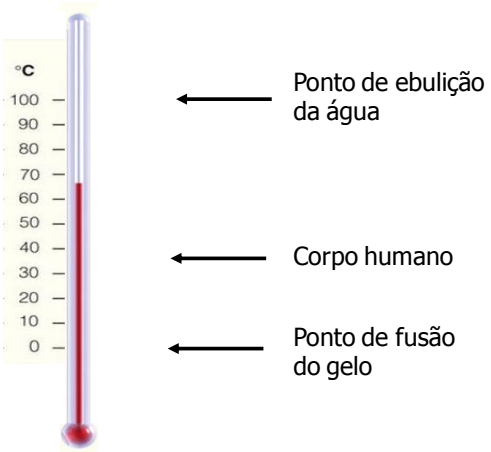


Figura 13: Um termômetro com escala

Existem perdas nessa transformação. O médico irá anotar o número correspondente ao tracinho mais próximo à extremidade da coluna, talvez com uma aproximação entre valores de dois tracinhos, mas não a sua altura exata. Para o médico essa perda pode ser perfeitamente tolerável; temperaturas do corpo humano medidas com uma casa decimal provavelmente atendem a todas as necessidades clínicas. Mas existe também um grande ganho: a temperatura anotada é informação simbólica, que pode ser comunicada por telefone, ou copiada em outro papel, ou digitada em um computador.

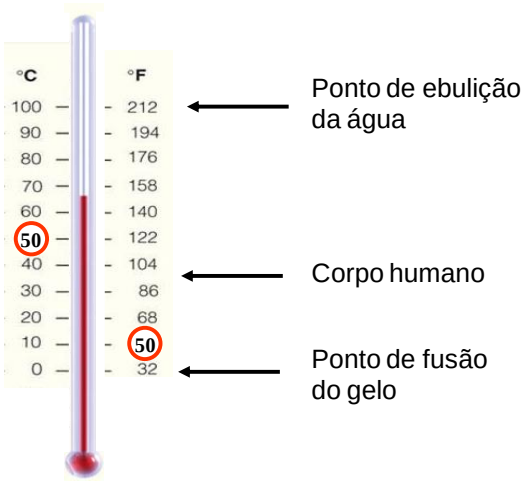


Figura 14: Termômetro com escalas Celsius e Farenheit

O uso de informação simbólica requer convenções de interpretação. A qual temperatura – a qual altura da coluna de mercúrio – corresponde o símbolo 50? A Figura 14 mostra escalas comumente utilizadas para temperaturas. Ou seja, ao dizer 50 graus, devemos esclarecer se estamos falando de graus centígrados, ou graus Farenheit, ou de alguma outra escala de temperaturas.

Tabela 1: Diferentes símbolos para os números inteiros de 1 a 7

Arábico	Romano	Binário
1	I	1
2	II	10
3	III	11
4	IV	100
5	V	101
6	VI	110
7	VII	111

Símbolos podem também ser usados para representar outros símbolos. Assim como explicitar uma escala é um requisito para se interpretar um número associado a uma temperatura, a correspondência entre símbolos deve ser estabelecida por convenções, como mostra o exemplo na Tabela 1.

1.3 Computadores e Informação

Computadores são essencialmente formados por *processadores*, *memórias*, *sensores* e *atuadores*:

- O componente mais importante é o processador. Um processador transforma informação exclusivamente simbólica, em informação também simbólica; as transformações que um processador realiza são ditadas por um *programa* que o processador executa.
- *Memórias* servem para registrar informações para recuperação posterior, e também só trabalham com informações simbólicas.
- *Sensores* como o teclado do computador, o mouse, câmeras e microfones digitais, entradas de rede, telas sensíveis ao toque, são também chamados de *dispositivos de entrada*, e trazem informação para o processador.

- *Atuadores* são impressoras, monitores de vídeo, alto-falantes, projetores, saídas de rede, e são também chamados de *dispositivos de saída*. Atuadores exportam informação que sai do processador.
- Sensores e atuadores frequentemente trabalham tanto com informação digital como com informação analógica, e fazem uso de conversões A/D e D/A.

A informação digital nos processadores e memórias atuais utiliza somente dois símbolos. A palavra *bit* designa a unidade de informação simbólica; os dois valores possíveis para um bit são normalmente denotados por 0 e 1.

Dois símbolos só? Computadores fazem maravilhas: exibem filmes, ajudam a projetar automóveis, controlam metrôs e aviões, oferecem jogos, editores de texto, correio eletrônico, fazem de tudo. Como, somente com 0 e 1? É que, mesmo se cada bit só contém um de dois símbolos, computadores usam muitos bits. Com 1 bit podemos representar dois estados, que podem por convenção representar 0 ou 1, sim ou não, verdadeiro ou falso, preto ou branco, verde ou amarelo, o que quer que se convencione, mas apenas dois estados. Com dois bits já são 4 combinações: 00, 01, 10 e 11. Com 3 bits, 8 combinações: 000, 001, 010, 011, 100, 101, 110 e 111. Já fica possível armazenar 8 diferentes informações, que poderiam ser os inteiros de 0 a 7, ou os inteiros entre -3 e 4, as letras entre A e H, ou talvez 8 diferentes níveis de cinza: o preto, o branco, e 6 nuanças intermediárias.

Não é difícil ver que, ao acrescentar um bit a um conjunto de bits, multiplicamos por 2 o número de combinações já existentes. Ou seja, com 1 bit, 2 estados; com 2 bits, $2 \times 2 = 4$ estados; com 3 bits, $2 \times 4 = 8$ estados. Generalizando, com n bits, temos 2^n combinações, e 2^n cresce muito rapidamente quando o valor de n aumenta:

- Um conjunto de 8 bits é chamado de *byte*. Com 8 bits podemos representar $2^8 = 256$ coisas diferentes. Isso é suficiente para atribuir um código distinto para cada letra do alfabeto, distinguindo entre maiúsculas e minúsculas, e também para caracteres especiais como "(", "+", etc.
- Com 24 bits, temos $2^{24} = 16.777.216$ possibilidades, o suficiente para representar todas as cores com qualidade excelente para a acuidade visual humana.
- Com 80 bits, nada menos que $2^{80} = 1.208.925.819.614.629.174.706.176$ coisas podem ser representadas!

A Tabela 2 mostra os prefixos usados para designar potências decimais e binárias de uso corrente na computação.

Tabela 2: Prefixos binários e decimais.

Prefixos Binários e Decimais			
Prefixo	**Símbolo**	**Valor**	**Base 10**
kilo	k/K	2^{10} = 1 024	> 10^{3} = 1,000
mega	M	2^{20} = 1 048 576	> 10^{6} = 1 000 000
giga	G	2^{30} = 1 073 741 824	> 10^{9} = 1 000 000 000
tera	T	2^{40} = 1 099 511 627 776	> 10^{12} = 1 000 000 000 000
peta	P	2^{50} = 1 125 899 906 842 624	> 10^{15} = 1 000 000 000 000 000
exa	E	2^{60} = 1 152 921 504 606 846 976	> 10^{18} = 1 000 000 000 000 000 000
zetta	Z	2^{70} = 1 180 591 620 717 411 303 424	> 10^{21} = 1 000 000 000 000 000 000 000
yotta	Y	2^{80} = 1 208 925 819 614 629 174 706 176	> 10^{24} = 1 000 000 000 000 000 000 000 000

Podemos ver que o uso de apenas 2 símbolos não traz limitação alguma de representatividade. Tudo bem, mas porque só 2 símbolos? Se os projetistas de computadores tivessem sido menos mesquinhos e usassem, por exemplo, os algarismos decimais como unidade básica de informação, não teríamos um sistema que, antes de qualquer coisa, seria familiar, e que com n algarismos poderíamos representar 10^n coisas diferentes?

A decisão de adoção de um sistema binário foi tomada pelos projetistas dos primeiros computadores, e se justifica principalmente pela confiabilidade. Computadores são

equipamentos eletrônicos, onde símbolos devem ser representados por voltagens ou correntes elétricas, ou por outras grandezas físicas. Se tivermos apenas dois níveis de voltagens ou de correntes, a distinção dos símbolos fica muito mais confiável.

Bits são baratos, e ocupam muito pouco espaço quando anotados em suportes eletrônicos. O notebook utilizado para escrever este texto em 2012 tinha 4GB (Giga Bytes) de memória principal, e 300GB de capacidade de armazenamento em sua memória secundária, um disco magnético.

1.4 Conversões análogo-digital e digital-analógica

Existem dispositivos que transformam informação analógica em informação digital (conversões A/D), e outros que fazem o contrário (conversões D/A). Muito frequentemente um fenômeno natural é usado para converter o fenômeno físico medido em impulsos elétricos, e estes impulsos são sinais de entrada usados na conversão análogo-digital.

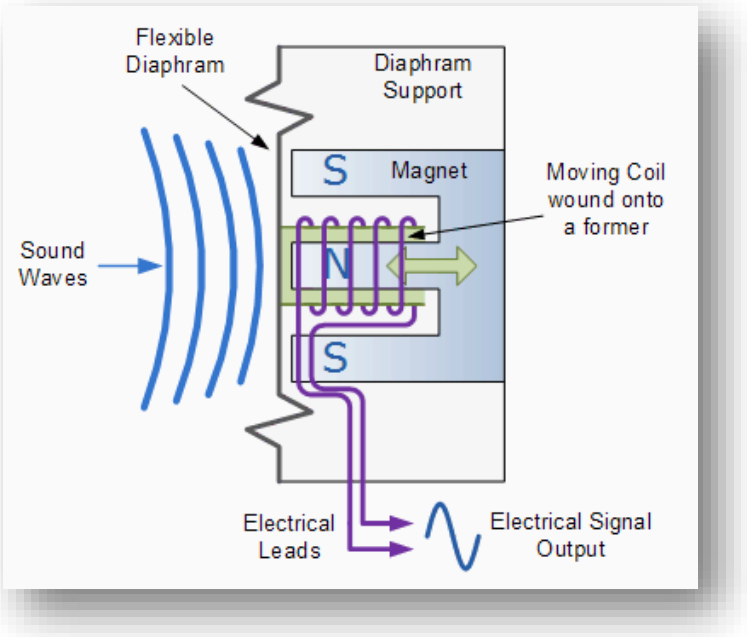


Figura 15: Um microfone converte ondas de pressão do ar em impulsos elétricos[22]

Um exemplo destes dispositivos é um microfone, que transforma sons em impulsos elétricos. Como mostra o esquema na Figura 15, as ondas de pressão no ar que constituem o som fazem o diafragma vibrar, arrastando consigo um cabo enrolado em solenóide. Este cabo se move em um campo magnético, gerando sinais elétricos proporcionais à deformação do diafragma. O sinal elétrico produzido pelo microfone pode então ser acoplado a um dispositivo de conversão análogo-digital.

Um alto-falante tem funcionamento similar: um impulso elétrico de entrada é aplicado sobre um cabo enrolado em solenoide, o que gera um campo magnético que faz mover uma membrana, que gera ondas de pressão no ar que constituem o som emitido.

[22] "Sound Transducer", acessado 2 de agosto de 2011, http://www.electronics-tutorials.ws/io/io_8.html.

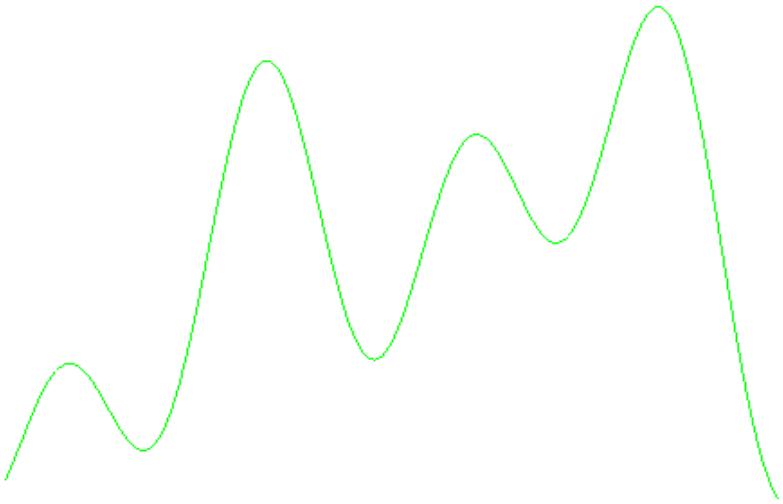


Figura 16: Um segundo de um sinal analógico

Para digitalizar um sinal de entrada como o da Figura 16 é preciso obter amostras de sua amplitude em instantes discretos no tempo, e digitalizar – obter um valor numérico – cada amostra.

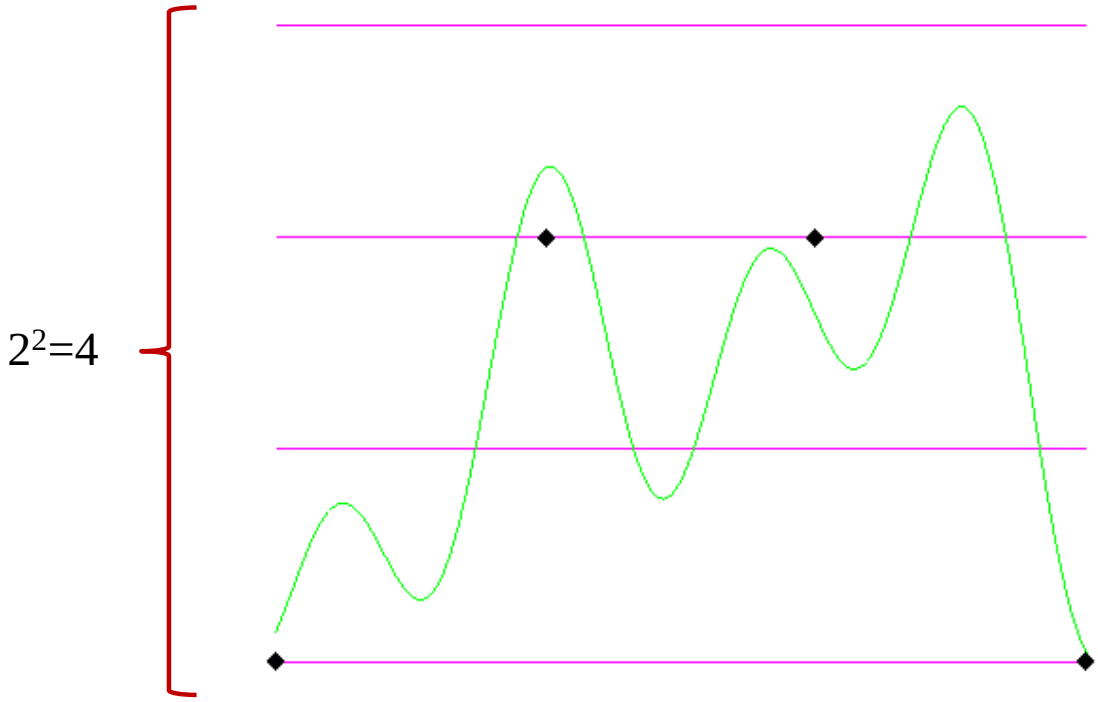


Figura 17: Sinal original e amostras, feitas 4 vezes por segundo, com precisão de 2 bits

A Figura 17 ilustra esse processo de digitalização de um sinal que varia ao longo do tempo. O resultado da digitalização é uma tabela – um conjunto de símbolos, com a informação mostrada na Figura 18.

Tempo(s)	Sinal(V)
0.00	00 (0)
0.33	10 (2)
0.66	10 (2)
1.00	00 (0)

Figura 18: Resultado da digitalização com 4 amostras/s e 2 bits de precisão

Após a digitalização, a informação analógica é perdida, e somente os símbolos resultantes da digitalização estão disponíveis. Para tentar reconstruir o sinal original a partir dos dados digitalizados, um primeiro passo poderia ser utilizar interpolação linear para obter dados intermediários entre os vazios existentes entre as amostras, como mostra a Figura 19

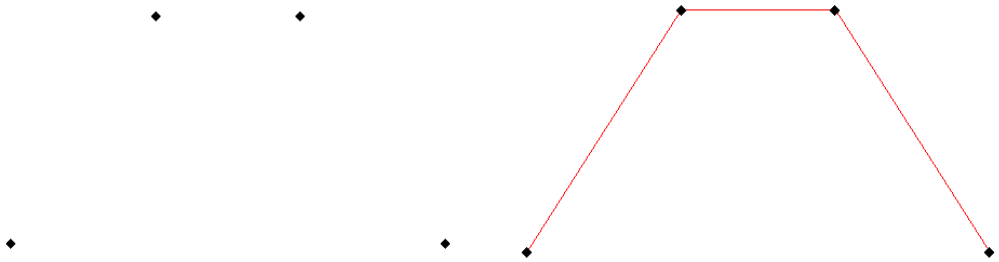


Figura 19: Reconstrução do sinal por interpolação linear

Se compararmos com o sinal original veremos que, para a nossa acuidade visual, o resultado não é nada bom, como mostra a Figura 20.

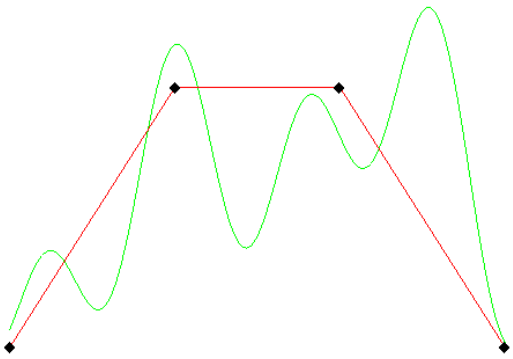


Figura 20: Sinal original e sinal reconstruído por interpolação linear a partir de digitalização com 4 amostras/s e 2 bits de precisão

Se tentarmos reconstruir o sinal com métodos mais sofisticados de interpolação, como *splines* (veremos mais tarde como utilizar splines), o resultado também não será bom, como mostra a Figura 21.

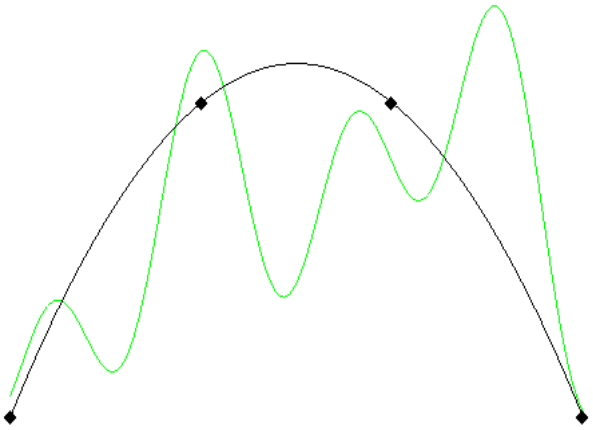


Figura 21: Sinal original e sinal reconstruído por interpolação por spline a partir de digitalização com 4 amostras/s e 2 bits de precisão

Para tentar melhorar a situação, vamos primeiramente aumentar a frequência de amostragem do sinal para 40 amostras/s, mantendo os 2 bits de precisão. A Figura 22 mostra as amostras obtidas pelo processo de digitalização, e a Figura 23 sua reconstrução por interpolação linear e spline.

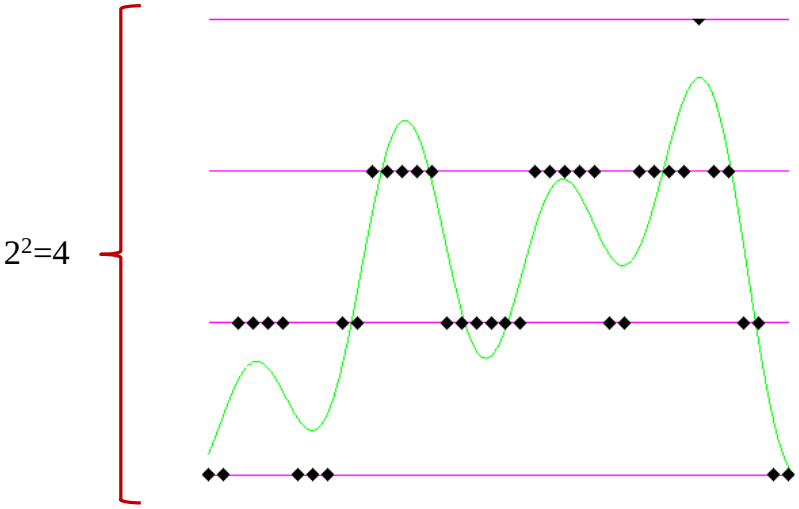


Figura 22: Sinal original e amostras, feitas 40 vezes por segundo, com precisão de 2 bits

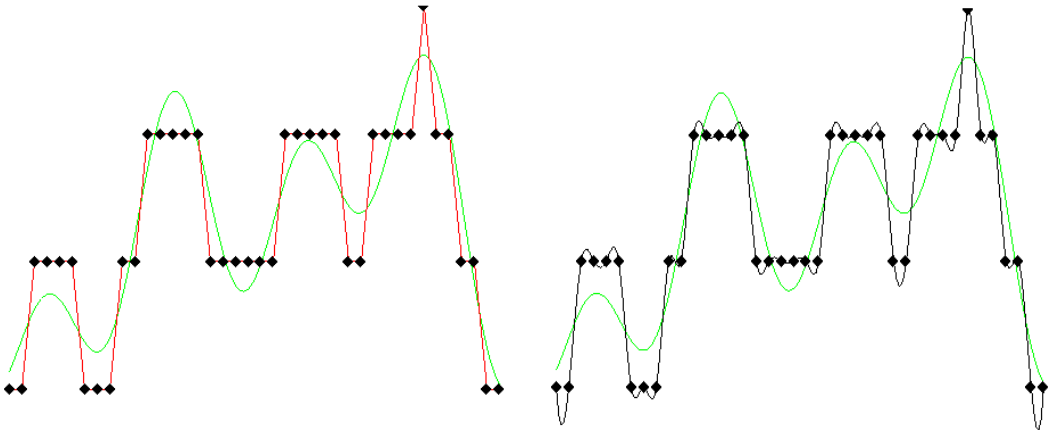


Figura 23: Sinal original e sinal reconstruído por interpolação linear e spline, com 40 amostras/s e 2 bits de precisão

Nós vemos que a reconstrução se aproximou mais do sinal original, mas o resultado ainda não é bom. É fácil ver que o problema agora está na precisão com que os valores são amostrados, pois com 2 bits só temos 4 possibilidades de atribuição de valores para as amostras.

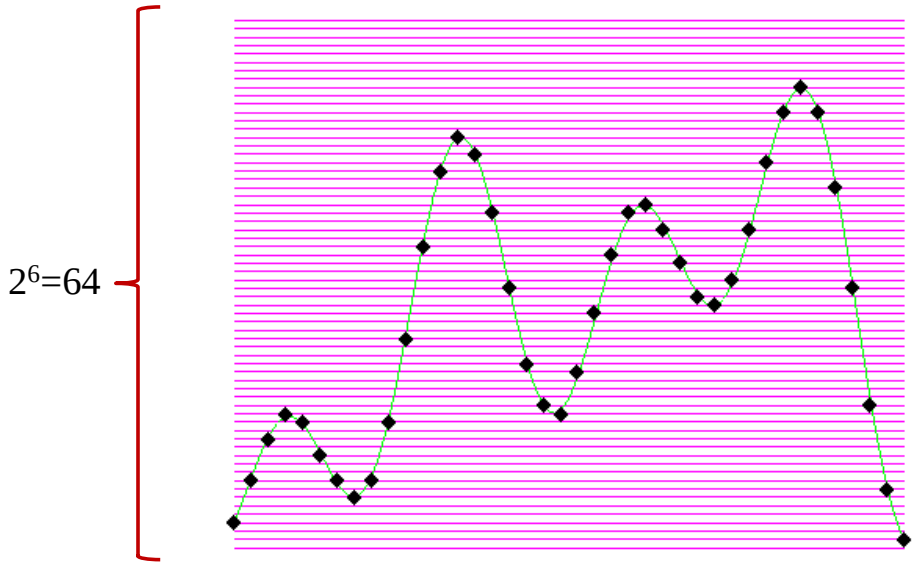


Figura 24: Sinal original e amostras, feitas 40 vezes por segundo, com precisão de 6 bits

Se utilizarmos 6 bits de precisão (ou 6 bits de *profundidade*), teremos $2^6 = 64$ valores possíveis para cada amostra, e a qualidade da digitalização é claramente superior, como mostra a Figura 24.

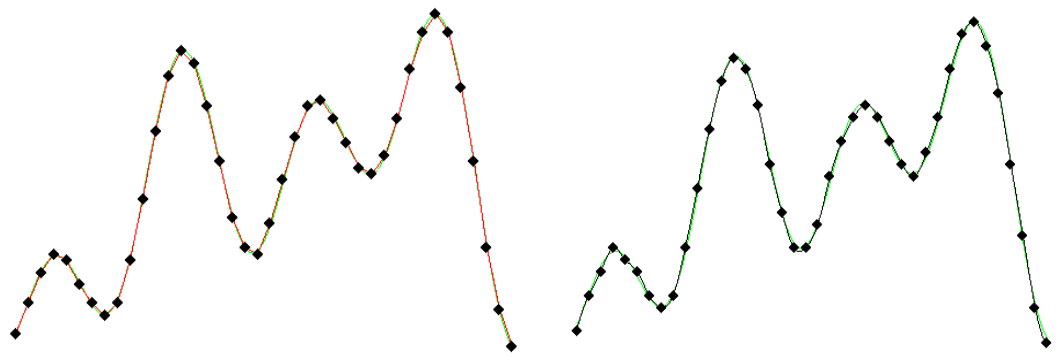


Figura 25: Sinal original e sinal reconstruído por interpolação linear e spline, com 40 amostras/s e 6 bits de precisão

Na reconstrução do sinal, como mostra a Figura 25, os resultados são muito bons com os dois métodos de interpolação.

Portanto, a qualidade da digitalização depende:

- da freqüência da amostragem e
- da precisão com que é feita cada amostragem.

Podemos sempre tentar aumentar a o número de bits e a freqüência de amostragem, mas isso tem custos. Dependendo da utilização final do sinal reconstruído, os custos com a melhoria de qualidade nem sempre são compensatórios. Se considerarmos que o sinal está sendo amostrado para apreciação visual por humanos, a partir de um certo ponto a qualidade da digitalização atingirá os nossos limites de acuidade visual ou sonora. Um CD musical é amostrado com frequência de 44,1 kHz e 16 bits de precisão, o que propicia uma qualidade bastante satisfatória; um sistema de áudio sofisticado utiliza amostras com 96 kHz e 24 bits de profundidade, o que satisfaz ouvidos exigentes de músicos profissionais.

Como conclusão, é importante observar que a conversão de informação em qualquer dos dois sentidos A/D ou D/A nunca é perfeita, mas em muitos casos pode ser tão boa quanto se necessite. Em conversões D/A (Digital/Analógica) destinadas à absorção por humanos, a exploração de limites fisiológicos como a acuidade visual ou auditiva é muito utilizada.

1.5 Sensores e atuadores

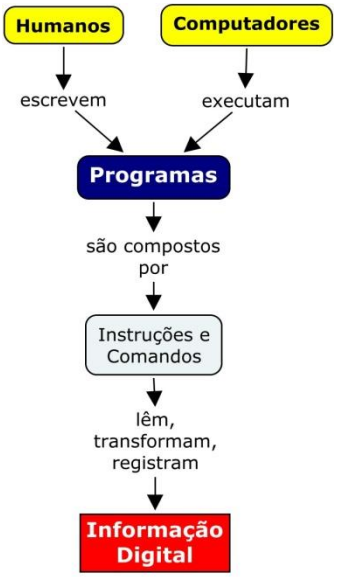


Figura 26: Computadores, programas e informação digital

Programas são escritos por pessoas e executados por computadores. Programas são compostos por instruções e comandos que lêm, transformam e registram informação digital.

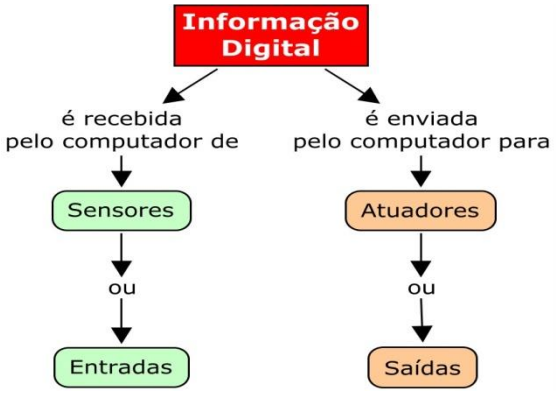


Figura 27: Sensores e atuadores

A informação digital é recebida pelo computador por sensores (ou equipamentos de entrada) e, normalmente após alguma transformação, é enviada para atuadores (ou equipamentos de saída).

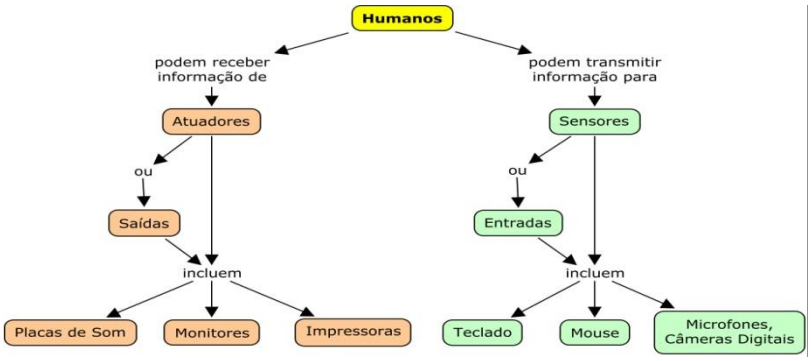


Figura 28: Interação homem-máquina

A interação entre humanos e computadores também se dá através de sensores e atuadores. Presente em quase todos os computadores, o sensor mais comum é o teclado. Sua importância é imensa. É através de teclados que entra, por exemplo, quase toda a informação que movimenta a economia mundial. O teclado pode ser também uma barreira pessoal para a entrada no mundo da informática. (Uma pessoa com dificuldades na digitação tende a abandonar os computadores, em uma atitude que provavelmente lhe trará conseqüências profissionais negativas.)

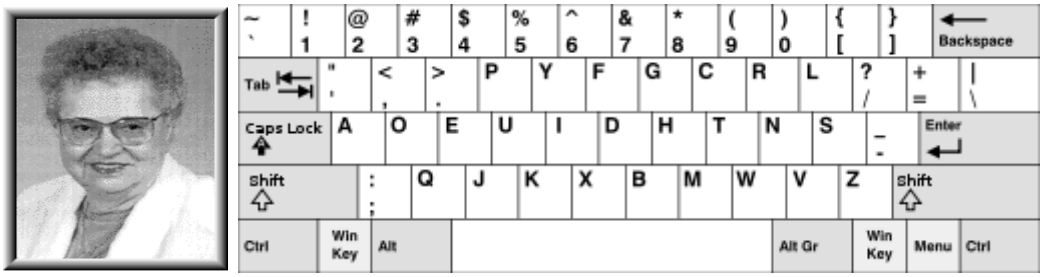


Figura 29: Barbara Blackburn, recordista mundial de velocidade de digitação[23], e um teclado Dvorak, projetado para maximizar a velocidade de digitação em inglês

Outras pessoas, pelo contrário, têm grande facilidade para a digitação. O recorde mundial de digitação pertence a Barbara Blackburn, que atingia uma velocidade sustentada de 15 toques por segundo, com picos de 20 toques por segundo, usando um teclado Dvorak (Figura 29).

[23] "Barbara Blackburn, the World's Fastest Typist", acessado 15 de setembro de 2012, http://rcranger.mysite.syr.edu/famhist/blackburn.htm.

Estamos nos permitindo essa digressão para calcular a velocidade de produção de informação por um teclado. Considerando que cada toque produz 8 bits (um byte), a Barbara Blackburn produzia informação a uma taxa máxima de 160 bps (bps = bits por segundo).

Figura 30: Uma camcorder de alta definição é um sensor que produz bits a uma taxa de 24 Mbps

Uma câmera de vídeo produz bits a taxas muito mais altas. Uma camcorder de alta definição pode gerar 24Mbps, ou 24 milhões de bits por segundo.

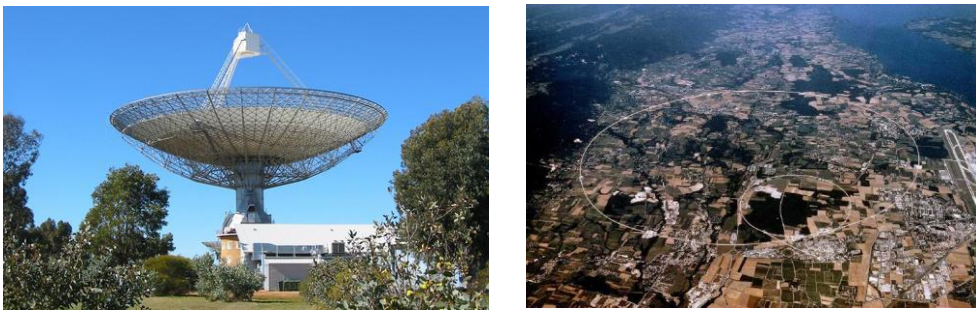


Figura 31: Sensores especiais: um rádio-telescópio e um acelerador de partículas

A ciência faz uso de sensores que produzem informação ainda muito mais rapidamente. Rádio-telescópios ou aceleradores de partículas podem produzir informação à taxas de Gbps, ou seja, bilhões de bits por segundo, milhões (literalmente) de vezes mais rápido do que o recorde mundial de digitação.

Figura 32: Um atuador especial: um braço mecânico

Atuadores comuns são monitores, impressoras, alto-falantes. Existem também atuadores especiais como braços mecânicos usados por robôs, como mostrado na Figura 32.

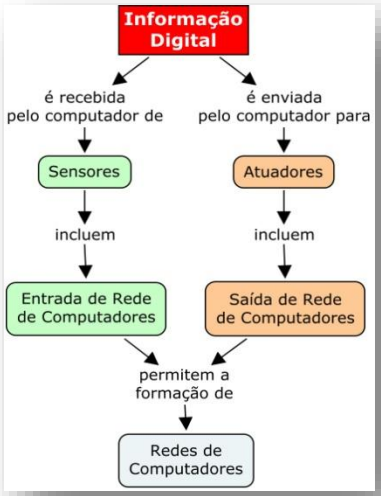


Figura 33: Redes de computadores

Entradas e saídas de rede são particularmente importantes pois permitem a conexão entre computadores. Um atuador de um computador emite sinais elétricos, ou de rádio, ou óticos, que são percebidos – lidos – por um sensor de outro computador.

1.6 Memórias

Memórias têm função análoga a papel e lápis: são usadas por computadores para registrar informações para recuperação posterior. Um computador geralmente trabalha com diversos tipos de memória, que seguem uma distribuição hierárquica:

- Registradores são memórias muito pequenas e muito rápidas, que se encontram dentro do mesmo chip do processador, e que têm suas entradas e saídas ligadas diretamente a circuitos que realizam transformações de informação, como a unidade aritmética que, como o nome indica, realiza operações aritméticas.
- Memória principal ou RAM (*Random Access Memory*) é um circuito externo ao processador, mas de acesso ainda bastante rápido. Instruções executadas pelo processador utilizam diretamente operandos armazenados na memória principal. Transformações como operações aritméticas geralmente exigem que informação seja previamente transferida da memória principal para registradores, onde as operações são realizadas, e os resultados posteriormente armazenados na memória principal. De uma forma geral a memória principal é *volátil*, no sentido em que é necessário manter o computador ligado para que a informação ali armazenada não se perca. A volatilidade não é uma necessidade, mas uma característica da tecnologia empregada nas memórias principais atuais. Há alguns anos atrás memórias principais utilizavam núcleos de ferrite, com o registro da informação feito por polarizações magnéticas não voláteis.

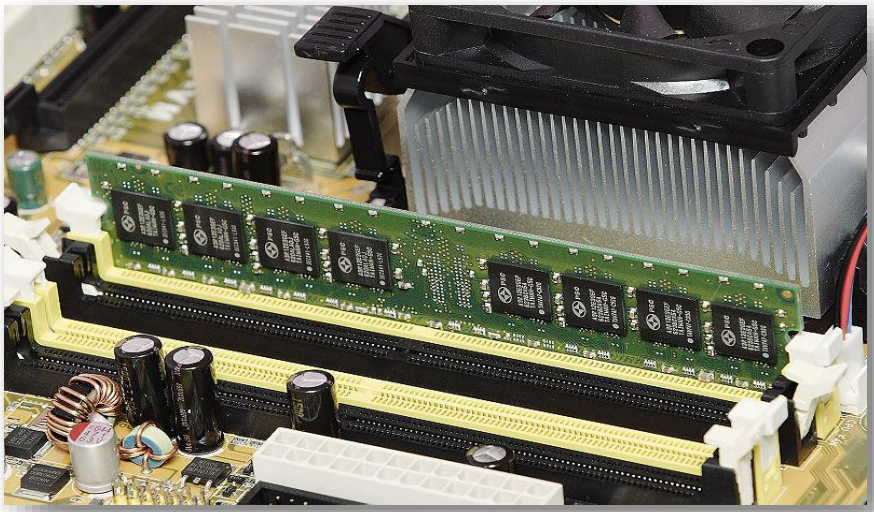


Figura 34: 4GB de RAM montados em um computador pessoal

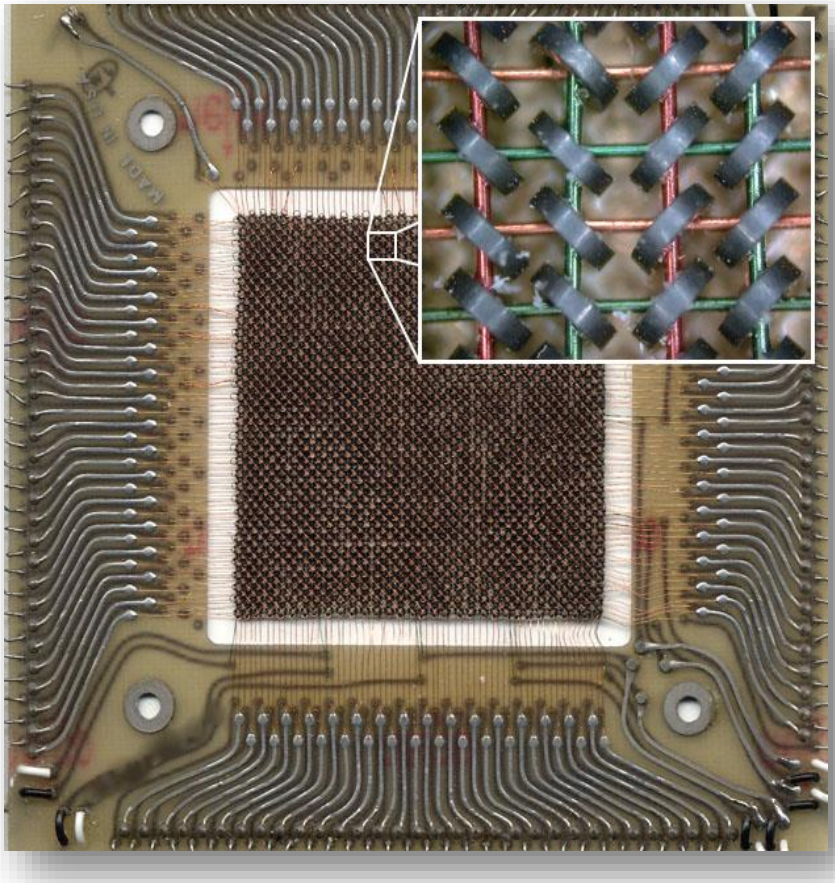


Figura 35: Memória de núcleos de ferrite[24], usada em 1964 pelo supercomputador CDC 6600, com 64 palavras de 64 bits em 11x11 cm

[24] "Magnetic-core memory - Wikipedia, the free encyclopedia", acessado 12 de março de 2011, http://en.wikipedia.org/wiki/Magnetic_core_memory.

- Memórias secundárias são tipicamente discos rígidos, onde informações também podem ser lidas e escritas, mas o processador deve executar instruções especiais de entrada e saída para isso. Memórias *flash* (memórias usadas em *pen drives*, no *iPod*) vêm sendo também cada vez mais utilizadas como memórias secundárias. Memórias secundárias são não-voláteis, com a informação armazenada permanecendo registrada mesmo sem qualquer alimentação de energia. A informação em uma memória secundária é quase sempre formatada em arquivos e diretórios, que provêm uma abstração essencial para o seu uso.

Figura 36: Um disco rígido sem a cobertura protetora, mostrando o movimento da cabeça de leitura e gravação

- Memórias terciárias são necessárias em ambientes maiores, que armazenam grandes volumes de dados. Fitas magnéticas são utilizadas, com um robô que é capaz de localizar a fita correta em um repositório e montá-la em uma unidade de fita ligada ao computador.

Figura 37: Armazenamento terciário com uma grande coleção de fitas magnéticas manipuladas por um braço robótico[25]

Na Figura 38 nós vemos uma forma de se "escrever" os símbolos 0 e 1 em um meio magnético, através de sequências de reversões ou não reversões de polarização de pequenas regiões gravadas em discos ou fitas.

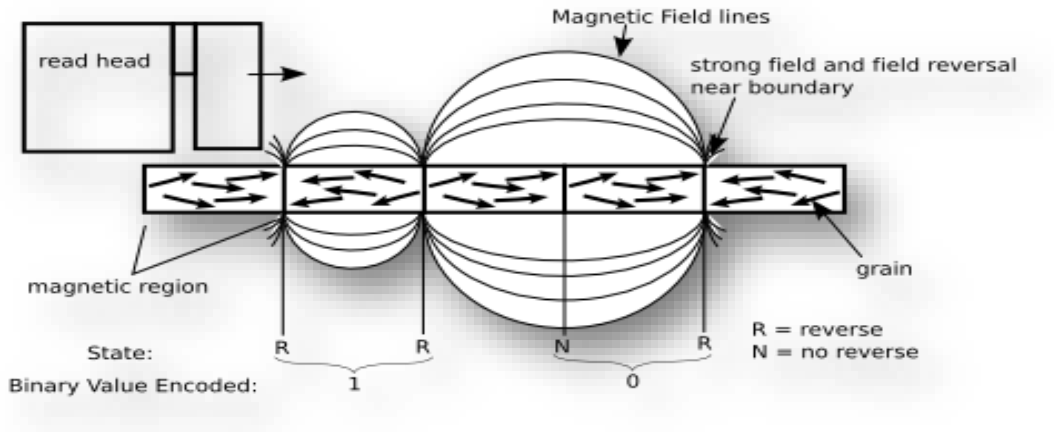


Figura 38: Escrevendo zeros e uns em meio magnético[26]

[25] "NCAR's MSS exceeds 2 petabytes", acessado 12 de agosto de 2011, http://www.cisl.ucar.edu/news/04/fotoweek/0729.mss2pb.html.
[26] "File:MagneticMedia.png - Wikipedia, the free encyclopedia", acessado 12 de agosto de 2011, http://en.wikipedia.org/wiki/File:MagneticMedia.png.

1.7 Organização do Conteúdo

Este material destina-se a disciplinas introdutórias de organização e programação de computadores para alunos de ciências e de engenharia, e tem como objetivos:

- introduzir noções básicas de circuitos digitais, organização de computadores, representação de dados e programação,
- apresentar ao aluno alguns princípios básicos da construção e da análise de algoritmos – métodos computacionais para transformação de informação – e de sua implementação em um ambiente de programação, e
- tornar o aluno fluente no uso de uma ferramenta computacional, o *Scilab*, de vasta aplicação nas ciências e engenharias.

O material está dividido em três partes:

- Parte I: Organização de Computadores
- Parte II: Ambiente e Linguagem Scilab
- Parte III: Algoritmos e Programas

A Parte I, Organização de Computadores, tem dois objetivos principais:

- dar ao aluno uma compreensão dos elementos essenciais do funcionamento interno de um computador, e
- permitir ao aluno perceber a dificuldade da programação em baixo nível, e apreciar os ganhos obtidos com o uso de compiladores e interpretadores

A interpretação de conjuntos de bits é estabelecida por convenções de códigos que associam a uma determinada configuração de bits um valor numérico, ou um nível de vermelho em uma imagem, ou o que quer que se convencione. São apresentados alguns dos códigos mais comumente utilizados na computação.

Circuitos combinatórios, isto é, circuitos digitais que realizam transformações sobre um conjunto de bits de entrada produzindo outro conjunto de bits como saída, não têm memória, e sua saída em um dado instante é função apenas dos valores de entrada nesse instante. Circuitos combinatórios utilizam portas lógicas, que são componentes que realizam as operações AND, OR e NOT que constituem a Álgebra de Boole. São vistos circuitos combinatórios para somas, comparações e para direcionamento de fluxo de dados.

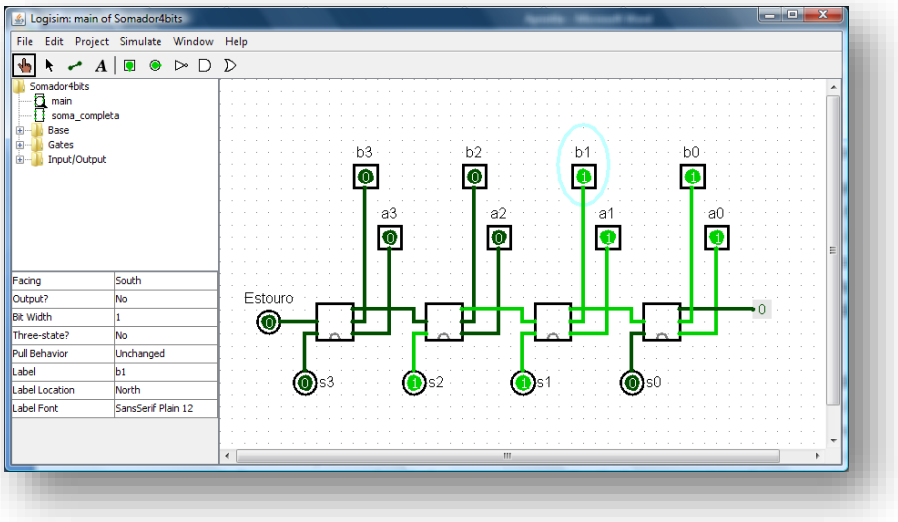


Figura 39: Circuito combinatório, simulado no Logisim, que realiza a soma de duas parcelas binárias de 4 bits

Circuitos seqüenciais têm a sua saída influenciada também pelo valor corrente de suas memórias. O elemento básico de memória é o flip-flop, capaz de armazenar um bit. Conjuntos de flip-flops formam registradores, que são ligados a outros registradores e a circuitos combinatórios por meio de barramentos. São também introduzidos circuitos de memórias onde ficam armazenados dados e programas. Osciladores e registradores circulares são introduzidos como elementos para controle da evolução temporal de um circuito.

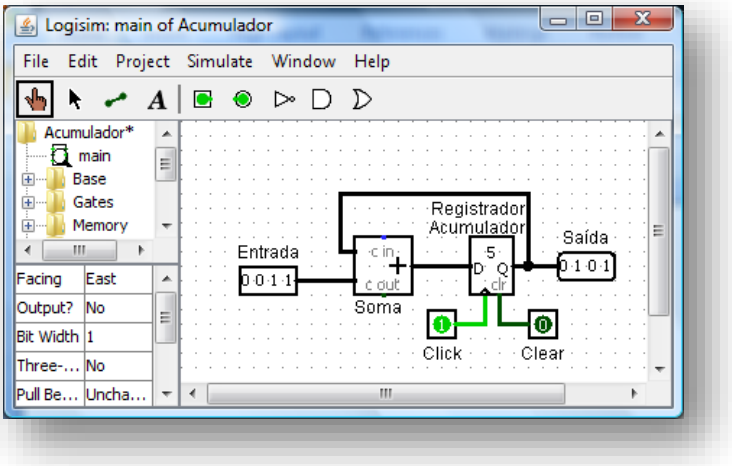


Figura 40: Circuito sequencial, simulado no Logisim, que acumula a soma das entradas.

A primeira parte culmina com a apresentação de processadores, que são circuitos que transformam informação de forma flexível, determinada por um programa que, por ser carregado em uma memória, pode ser substituído com facilidade. O estudo da organização de computadores terá permitido ao aluno tanto conhecer melhor o funcionamento básico de um computador, como ter um contato com as dificuldades da programação a nível de instruções de máquina de um computador.

A Parte II, Linguagem e Ambiente Scilab, tem como objetivos principais:

- a introdução de uma linguagem de alto nível, Scilab (Scilab Consortium, s.d.), que facilita imensamente a tarefa de programação de computadores através da oferta de comandos com formato muito mais próximo da forma como seres humanos raciocinam, e
- a familiarização do aluno com o ambiente de desenvolvimento e de execução de programas fornecido pelo Scilab.

Inicialmente é apresentado o Scilab como uma linguagem de alto nível, com variáveis, expressões aritméticas e comandos de atribuição. São vistas variáveis que contêm valores numéricos, lógicos ou cadeias de caracteres. Em seguida são apresentadas construções de linguagem para expressar comportamentos condicionais e repetitivos.

Matrizes constituem o ponto forte do Scilab. Nós veremos como criar e modificar matrizes, realizar operações de aritmética matricial, como construir matrizes a partir de matrizes já existentes, e uma série de outras operações. Um uso freqüente de matrizes no Scilab é para a construção de gráficos, feita por comandos muito flexíveis.

Em seguida são vistos comandos para a manipulação de arquivos, que são conjuntos de dados que tipicamente são produzidos por um programa e armazenados em um disco rígido ou um pen drive, e lidos posteriormente por outro programa. Arquivos são absolutamente essenciais para o tratamento de grandes volumes de dados.

A Parte II se encerra com o estudo de funções Scilab, que constituem uma ferramenta essencial para o uso de módulos na construção de programas.

Para exemplificar o uso do Scilab para a construção de programas, consideremos o seguinte problema. Temos um arquivo **`ondas.txt`** (fonte: (StatLib, 1989))que contém dados obtidos em um laboratório de observação de ondas marinhas (Figura 41).

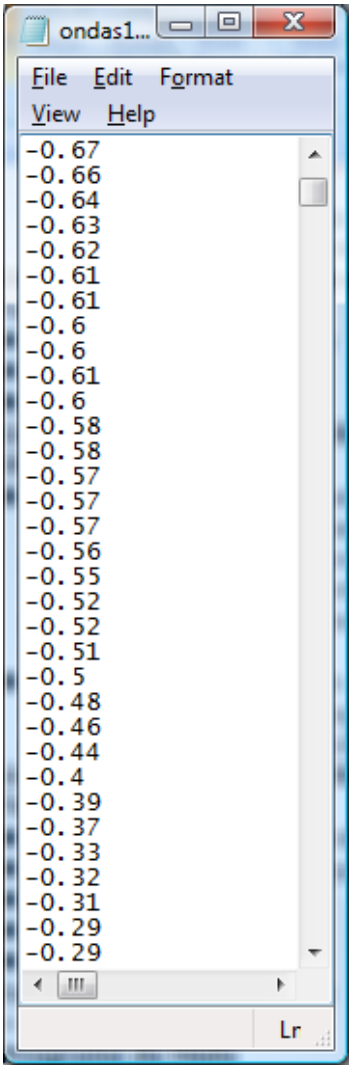


Figura 41: O arquivo ondas1.txt visto com o Bloco de Notas

Cada linha do arquivo contém uma medida do nível do mar; a aparelhagem do laboratório permite fazer 30 medidas por segundo. O arquivo completo tem 4096 linhas. Nós queremos um programa para obter um gráfico com os pontos do arquivo, e também saber os valores máximo e mínimo presentes na amostra.

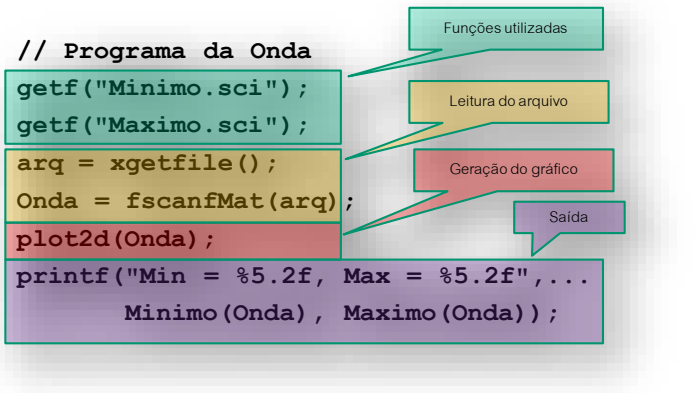


Figura 42: O programa Scilab Onda.sci. Cada linha é um comando, a não ser que termine em "...". Linhas que se iniciam com "//" são comentários que se destinam a leitores humanos.

A Figura 42 mostra um programa Scilab que faz isso, utilizando dois módulos, as funções **`Maximo`** e **`Minimo`**, mostradas na Figura 43.

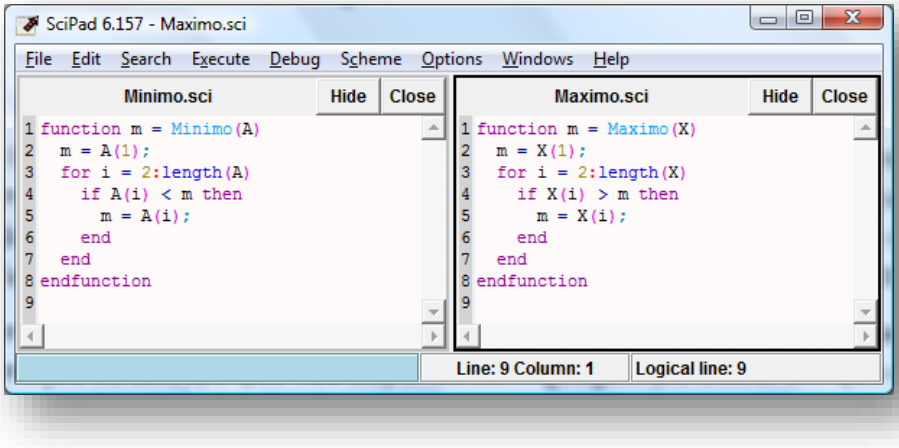


Figura 43: As funções Maximo e Minimo vistas no editor SciNotes

Para entender melhor essas funções, veja o algoritmo descrito a seguir nesta seção. O programa produz o gráfico da Figura 44, e a saída "`Min = -0.86, Max =  0.36`" mostrada na console do Scilab.

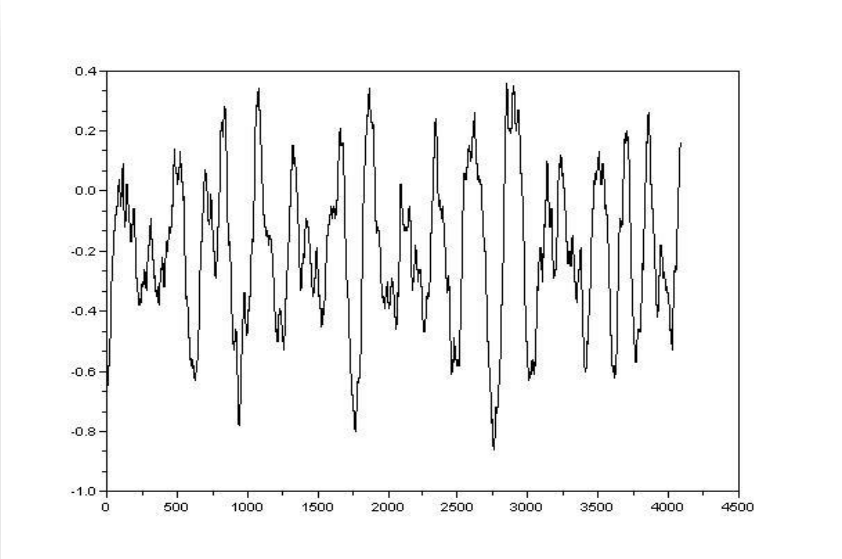


Figura 44: Gráfico com os pontos no arquivo ondas1.txt

Na parte III, Algoritmos, nosso foco se afasta um pouco de especificidades da linguagem Scilab para abordar aspectos da programação de computadores que são independentes da linguagem utilizada. Nós continuamos a usar o Scilab, mas o conteúdo dessa parte seria essencialmente o mesmo se estivéssemos utilizando C, Java, Fortran ou qualquer outra linguagem.

Nós veremos que um algoritmo é um método para transformação de informação de entrada em informação de saída, e que em boa parte é independente da linguagem particular em que está descrito. A transformação desejada é definida por uma especificação que resulta de uma análise de um problema.

```
A = [39. 24. 50. 42. 28. 8. 62. 34. 70.  52.];
```

Figura 45: Como encontrar o menor valor em uma coleção de números?

Como um exemplo, queremos encontrar o menor valor presente em uma coleção de números. Com 10 números, como mostrados na Figura 45, é fácil; podemos fazê-lo por inspeção visual. Mas como encontrar o menor valor entre, digamos, 50.000 números? Para isso precisamos de um *algoritmo*, isto é, de um método que tenha como informação de entrada a coleção de números e que, pela aplicação de uma sequência de operações, produza como saída o menor valor ali presente.

Em programação uma coleção de números é chamada *vetor*. O primeiro elemento de um vetor A é notado $A(1)$; o quinto, $A(5)$; o n-ésimo, $A(n)$. Queremos portanto construir um algoritmo que encontre o menor valor em um vetor de tamanho arbitrário. Para isso, vamos começar com o caso mais simples possível: um vetor com um único elemento, que é obviamente o menor elemento presente no vetor. Tudo bem, mas queremos trabalhar com vetores grandes de verdade.

Para avançar, usamos indução. Suponhamos que, de alguma forma, descobrimos que o menor valor entre os k primeiros elementos de um vetor de tamanho n é m. Podemos então inferir que o menor valor entre os $k+1$ primeiros elementos de A é o menor entre m e $A(k+1)$. Como já sabemos encontrar o mínimo em um vetor de 1 elemento ($k = 1$), sabemos encontrar o mínimo em um vetor com 2 elementos; como sabemos encontrar o mínimo em

um vetor de 2 elementos, sabemos encontrar o mínimo em um vetor com 3 elementos. Prosseguindo com o raciocínio, já temos um algoritmo para encontrar o menor valor em um vetor de tamanho arbitrário. Tendo um algoritmo, não temos mais medo do problema com 50.000 números. Nosso trabalho será programar o algoritmo; caberá ao computador a sua execução.

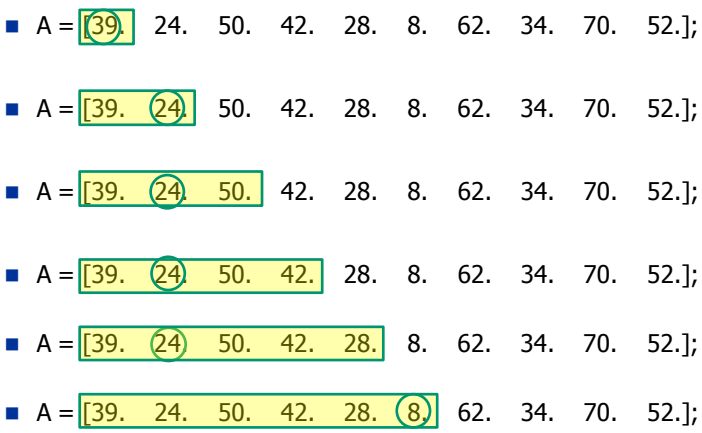


Figura 46: Seis primeiros passos do algoritmo que encontra o menor valor presente em um vetor

Um algoritmo é dito correto quando atende à sua especificação. Dois algoritmos corretos podem satisfazer uma mesma especificação, mas diferir substancialmente na eficiência (gasto de tempo e de recursos computacionais como memória) com que realizam a transformação de informação desejada. O termo *complexidade* é empregado para designar a eficiência de um algoritmo para a solução de um problema.

A complexidade de algoritmos é estudada inicialmente com dois problemas clássicos da ciência da computação: a ordenação dos elementos de um vetor e a busca por um elemento de um vetor com valor igual a uma chave dada. Para cada um desses problemas são vistos algoritmos que diferem em sua complexidade.

Encontrar uma raiz de uma função ou resolver um sistema de equações lineares são exemplos de problemas numéricos que um cientista ou engenheiro frequentemente tem que resolver. Nós veremos alguns exemplos de algoritmos que resolvem problemas desse tipo, e também alguns cuidados que devem ser tomados ao se desenvolver programas para estas aplicações.

Algoritmos com melhor complexidade são fruto de engenho e arte de seus projetistas. Existem entretanto problemas cuja solução algorítmica é intrinsecamente difícil, no sentido em que não existem boas soluções para eles. A Parte III e o curso se encerram com exemplos de problemas computacionalmente difíceis, e de problemas para os quais simplesmente não existe nenhum algoritmo que os resolvam para todos os casos.

2 Circuitos Combinatórios

Computadores transformam informação de entrada em informação de saída. Entradas e saídas podem lidar com informação analógica, mas um processador, o principal componente dos computadores, transforma informação exclusivamente simbólica de entrada em informação simbólica de saída. Isto é feito por composições de transformações básicas extremamente simples sobre os símbolos 0 e 1. Neste módulo nós veremos quais são essas transformações básicas e como essas transformações básicas são implantadas com transistores, o que permite a sua implementação em semicondutores com dimensões físicas muito reduzidas. Veremos também como combinar essas transformações para obter circuitos que realizam a soma ou a comparação de dois números, ou o estabelecimento de conexões flexíveis, guiadas por informações de endereço, para o transporte de informação em um circuito lógico.

2.1 Bits e códigos

Para ter utilidade, informação simbólica exige convenções de interpretação. Qualquer pessoa pode dar a um conjunto de bits a interpretação que bem entender, e programadores com freqüência escolhem codificações que atendem às suas próprias necessidades, sem outras preocupações. Mas quando se quer enviar ou receber dados de outras pessoas, ou utilizar circuitos já construídos para a realização de operações aritméticas, a escolha de um código com maior aceitação é praticamente uma imposição. Nós veremos a seguir alguns dos códigos mais utilizados em computação.

ASCII e UNICODE. Para a comunicação com humanos a linguagem escrita é essencial. Seres humanos lêm e escrevem usando algum alfabeto, algarismos, símbolos de pontuação, e esta é uma parte essencial de nossa comunicação. O código ASCII – *American Standard Code for Information Interchange* – é uma dessas convenções de ampla aceitação em toda a indústria da computação, e que tem como objetivo a codificação de símbolos úteis, chamados *caracteres*, para a comunicação por escrito com humanos.

Caractere	Código	Decimal
End of Transmission	0000100	4
Line Feed	0001010	10
Space	0100000	32
(	0101000	40
+	0101011	43
0	0110000	48
1	0110001	49
2	0110010	50
3	0110011	51
A	1000001	65
B	1000010	66
C	1000011	67
a	1100001	97
b	1100010	98
c	1100011	99

Figura 47: Exemplos de codificação ASCII

A Figura 47 mostra partes do código ASCII, cuja primeira versão foi proposta em 1963[27]. O código foi projetado para a representação de caracteres como “A”, “a”, “(“, “+”, etc., para o espaço em branco (sim, o espaço também exige uma representação), e para os chamados caracteres de controle, como *line feed*, para indicar uma troca de linhas. O código ASCII oficial usa 7 bits, o que permite $2^7 = 128$ combinações. Destas, 33 representam caracteres de controle como *Line Feed* ou *Carriage Return*, muito úteis, especialmente nos tempos em que teletipos – máquinas de escrever comandadas por computador – eram um periférico essencial para os operadores.

Figura 48: Dennis Ritchie (em pé) e Ken Thompson, fotografados durante o desenvolvimento do sistema operacional Unix (o Linux é uma versão do Unix, (re)escrita por Linus Torvalds) e da linguagem C, usando um teletipo como terminal de um PDP-11, em 1972[28]

Tendo sido proposto por norte-americanos no início da era dos computadores, não é de se estranhar que no código ASCII não haja provisão para caracteres acentuados ou cedilhas. Estes são contemplados no código ASCII estendido, que usa 8 bits (um byte) para representar 256 caracteres.

O sucessor moderno do código ASCII é o Unicode[29], um padrão de codificação capaz de representar caracteres chineses, árabes, tailandeses, enfim, de praticamente qualquer conjunto de caracteres das línguas vivas e mortas do mundo. Unicode permite diversas codificações em bits. A codificação mais importante – a mais utilizada na Internet – é a UTF-8, que utiliza entre 1 e 4 bytes para a representação de caracteres. Os caracteres representados

[27] “ASCII - Wikipedia, the free encyclopedia”, acessado 18 de agosto de 2011, http://en.wikipedia.org/wiki/ASCII.
[28] “Ken&Den picture”, acessado 24 de setembro de 2012, http://cm.bell-labs.com/cm/cs/who/dmr/picture.html.
[29] “Unicode - Wikipedia, the free encyclopedia”, acessado 19 de agosto de 2011, http://en.wikipedia.org/wiki/Unicode.

por um único byte têm a mesma representação em UTF-8 e em ASCII, uma compatibilidade essencial para o bom funcionamento de programas que contavam com codificação ASCII.

Binários sem Sinal. Um outro critério para a escolha de um código é a sua adequação para operações aritméticas. No código ASCII estendido o número decimal 123 pode ser representado pela seqüência de 24 bits 00110001 00110010 00110011 (colocamos espaços para facilitar a leitura). Esta é uma codificação muito boa para a impressão de números por dispositivos que permitem leitura por humanos, e também para a codificação da entrada produzida por teclados. Mas para operações aritméticas, a interpretação de um conjunto de bits como um número em notação posicional na base 2 leva a códigos mais compactos e que, como veremos, permitem a realização de operações de soma por circuitos mais simples, mais rápidos e mais baratos.

n	0	1	2	3	4	5	6	7	8	9	10
2^n	1	2	4	8	16	32	64	128	256	512	1024

Figura 49: Tabela com potências de 2

O decimal 123 pode ser representado em binário por 1111011, utilizando apenas 7 bits. Esta representação vem do fato que $123 = 1.2^6 + 1.2^5 + 1.2^4 + 1.2^3 + 0.2^2 + 1.2^1 + 1.2^0$.

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binário	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Figura 50: Representação dos números de 0 a 15 como binários sem sinal de 4 bits

A conversão de 123_{10} para 1111011_2 pode ser feita com o seguinte método para conversão binário-decimal:

- Nós precisamos expressar 123 como uma soma de potências de 2. Para isso, procuramos em uma tabela de potências de 2 (como a tabela na Figura 49, mas talvez maior, pois a tabela deve chegar até uma potência de 2 maior que o número a ser convertido) o maior valor que não exceda 123 o que, no caso, é 64. Temos $123 = 64 + (123 - 64) = 64 + 59$.
- Temos agora que expressar 59 como uma soma de potências de 2. Na tabela da Figura 49, a maior potência de 2 que não excede 59 é 32. Repetindo o raciocínio, temos $59 = 32 + (59 - 32) = 32 + 27$.

Desta forma progredimos até que a parcela restante a converter como soma de potências de 2 seja igual a zero.

A converter	123	59	27	11	3	1	0
Potência de 2	64	32	16	8	2	1	

Figura 51: Conversão do decimal 123 para binário

Um uso comum de números representados como binários sem sinal é o sistema RGB (*Red*, *Green*, *Blue*) para a codificação da cor associada a um pixel – um ponto em um monitor – com os valores dos níveis de intensidade das cores primárias componentes vermelho, verde e azul. Para cada uma dessas cores utiliza-se um byte (8 bits), o que permite representar níveis de intensidade entre 0 e 255. Essa precisão é considerada satisfatória para a nossa acuidade visual.

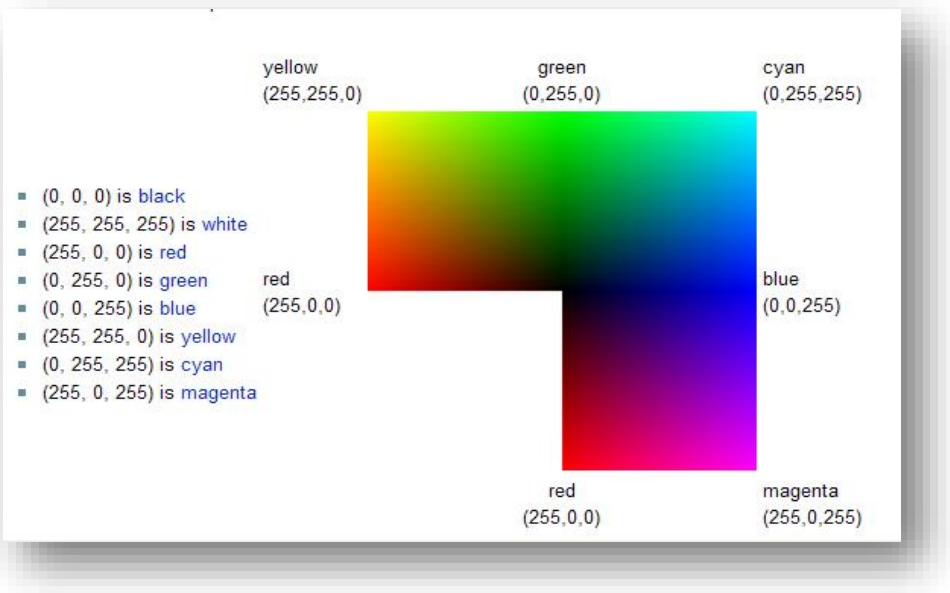


Figura 52: Uso de binários sem sinal para a representação dos níveis de intensidade das cores primárias vermelho (Red), verde (Green) e azul (Blue), conhecido como RGB

Sinal e Magnitude. Como computadores só usam bits, nós não podemos usar um sinal "-" para indicar que um valor é negativo. Temos que usar bits para codificar essa informação. A codificação para números negativos conhecida como *sinal e magnitude* é bastante natural. Basta tomar o bit mais à esquerda e interpretá-lo como o sinal do número: se for 0, o número cuja amplitude é representada pelos bits restantes é positivo, e se for 1, negativo. Portanto, se tivermos 8 bits, o bit mais à esquerda será usado para codificar o sinal. Os 7 bits restantes nos permitem representar amplitudes entre 0 e 127, ou seja, podemos, com os 8 bits, representar os inteiros entre -127 e +127. Repare que zero tem duas representações: 10000000 e 00000000.

Codificação com Deslocamento. Uma outra possibilidade para representar números negativos está ilustrada na Figura 53. A convenção adotada é de interpretar um conjunto de bits como a representação de um valor igual ao valor de sua interpretação como binário sem sinal, deslocado por um fator a ser subtraído.

Decimal	Binário	Número
0	000	-3
1	001	-2
2	010	-1
3	011	0
4	100	1
5	101	2
6	110	3
7	111	4

Figura 53: Representação de números negativos por deslocamento

Complemento de 2. A absoluta maioria dos computadores utiliza uma codificação conhecida como complemento de 2 para a representação de números inteiros negativos. A Figura 54 mostra a codificação em 3 bits dos inteiros entre -4 e 3 usando complemento de 2.

Complemento de 2			
Bits			Valor
b_2	b_1	b_0	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	-4
1	0	1	-3
1	1	0	-2
1	1	1	-1

Figura 54: Codificação em complemento de 2 com 3 bits

O valor representado pela sequência de bits $b_2b_1b_0$ é dado pela fórmula

$$V = -4b_2 + 2b_1 + b_0.$$

(Nós adotaremos neste texto a convenção de designar sempre por b_0 o bit menos significativo e por b_n o bit mais significativo de uma sequência b de n bits.) O valor representado por 101 é, portanto, dado por $V = -4.1 + 2.0 + 1.1 = -3$.

Você pode estar se perguntando como é que alguém pode preferir usar complemento de 2, sendo que as propostas anteriores são muito mais simples e intuitivas. A motivação é simples, e decisiva para todos os fabricantes de computadores: custo. A representação em complemento de 2 tem propriedades que permitem uma grande economia na construção de circuitos. Para se obter o negativo de um número, basta complementá-lo bit a bit, e somar 1. Por exemplo, 2 é representado por 010; complementando bit a bit, obtemos 101; somando 1, chegamos a 110, que é a representação de -2. Ao inverter todos os bits de 000, obtemos 111; somando 1, obtemos 1000, caso em que só os três bits menos significativos devem ser aproveitados. Isso permite aos fabricantes aproveitar para fazer subtrações usando o mesmo circuito utilizado para fazer somas, com acréscimos mínimos de hardware.

Ponto Flutuante. A representação em um número limitado de bits de valores muito grandes ou muito pequenos utiliza mecanismos para sua codificação similares aos que usamos na notação científica com potências de 10. O número de Avogadro, por exemplo, é notado por $6{,}02 \times 10^{23}$, e o diâmetro de um átomo de hidrogênio é de $1{,}0 \times 10^{-8}$ metros. Para o número de Avogadro nós dizemos que a *mantissa* ou *significando* é 6,02 e o *expoente*, 23. Este mesmo número poderia ser escrito como 602×10^{21}, ou como $0{,}000602 \times 10^{27}$, mas é costume adotar a forma *normalizada*, isto é, com um único algarismo não nulo antes da vírgula.

A codificação de números "em ponto flutuante" atribui a alguns dos bits do número o significado de um expoente não de 10 mas de 2. Existe um padrão para ponto flutuante de aceitação total pela indústria que é o IEEE 754. Para números de precisão simples, o IEEE 754 utiliza 32 bits, sendo 1 bit para o sinal (0 = positivo, 1 = negativo), 8 para o expoente e 23 para a mantissa. Para precisão dupla são 64 bits: 1 para o sinal, 11 para o expoente e 52 para a mantissa. A codificação por deslocamento que acabamos de ver é utilizada para permitir a representação de expoentes negativos. Para número de 32 bits, o deslocamento do expoente é de 127; para números de 64 bits, de 1023. A mantissa é normalizada, isto é, tem um único algarismo não nulo antes da vírgula – que, como é sempre igual a 1 no sistema binário, não necessita ser representado.

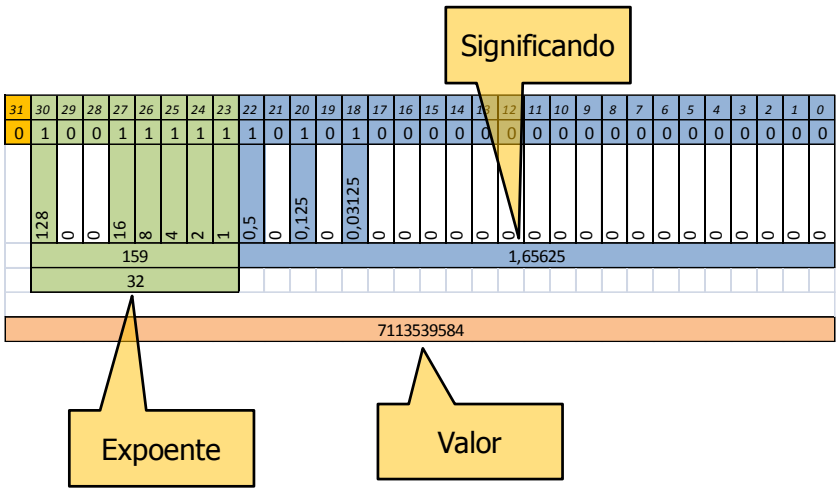


Figura 55: Um número em ponto flutuante com 32 bits

A Figura 55 mostra um número em ponto flutuante de 32 bits. Você deve reparar que os bits do expoente codificam 159, se interpretados como um binário sem sinal; com o deslocamento, o expoente de 2 é igual a $159 - 127 = 32$. O número ali codificado é portanto igual a

$$(-1)^0 \times (1 + 2^{-1} + 2^{-3} + 2^{-5}) \times 2^{32} = 1{,}65625 \times 2^{32} = 7113539584$$

O padrão IEEE 754 define interpretações especiais para expoentes só com 1s e só com 0s, incluindo o próprio zero (com expoente e mantissa iguais a zero), valores especiais como $\pm\infty$ ou NAN (*Not A Number*), que são utilizados para a comunicação de resultados de certas operações ou do cálculo de algumas funções. A referência [30] tem um bom resumo do padrão IEEE 754.

2.2 Álgebra de Boole

Em 1854 o matemático inglês George Boole propôs uma álgebra para o cálculo da validade de expressões formadas por proposições lógicas[31]. Quase um século depois, Claude Shannon[32], então com 21 anos, em sua dissertação de mestrado defendida no MIT em 1937, demonstrou que o projeto de circuitos lógicos, que viriam a constituir a base dos computadores, poderia se beneficiar diretamente destes resultados. Shannon tinha tomado conhecimento dos trabalhos de Boole em um curso de filosofia. Essa álgebra é chamada hoje de Álgebra de Boole, e constitui a base para o projeto de circuitos digitais. Ela trabalha com variáveis lógicas, isto é, com variáveis que podem ter somente os valores *verdadeiro* ou *falso*, ou 1 ou 0.

[30] Steve Hollasch, "IEEE Standard 754 Floating-Point", acessado 7 de março de 2011, http://steve.hollasch.net/cgindex/coding/ieeefloat.html.

[31] "An investigation of the laws of thought [microform] : on which are founded the mathematical theories of logic and probabilities : Boole, George, 1815-1864 : Free Download & Streaming : Internet Archive", acessado 19 de agosto de 2011, http://www.archive.org/details/investigationofl00boolrich.

[32] "Claude Shannon - Wikipedia, the free encyclopedia", acessado 19 de agosto de 2011, http://en.wikipedia.org/wiki/Claude_Shannon.

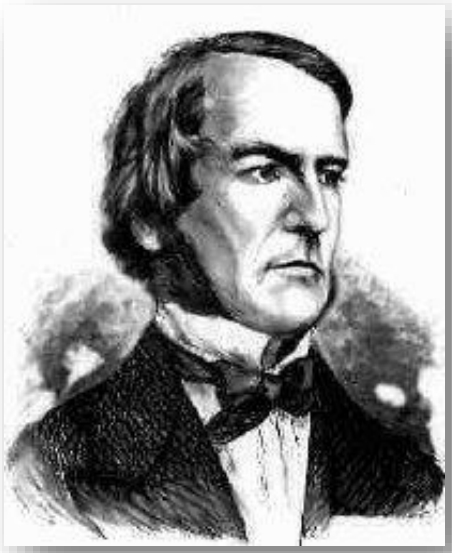
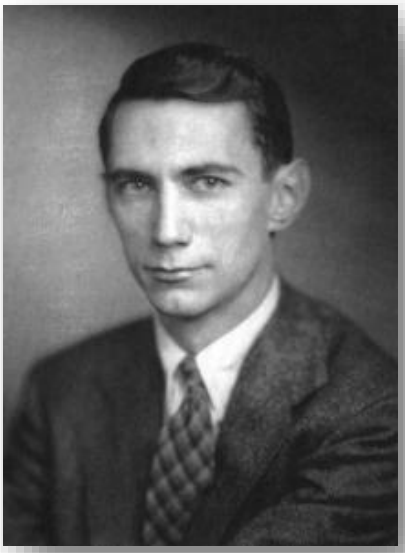



Figura 56: George Boole, 1820-1898[33], e Claude Shannon, 1916-2001[34]

As operações fundamentais da Álgebra de Boole são **NÃO**, **E** e **OU**, mais conhecidas pelas palavras em inglês **NOT**, **AND** e **OR**. Essas operações fundamentais, ou melhor, composições dessas operações fundamentais constituem as únicas transformações feitas por qualquer computador sobre bits.

a	NOT a
0	1
1	0

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

Figura 57: Definição das operações NOT, OR e AND

A Figura 57 mostra as definições dessas operações. O que se vê nas tabelas é bastante intuitivo:

- A operação de negação (**NOT**) é uma função de um único argumento, e simplesmente inverte o valor de entrada;
- A operação **OR** é uma função de 2 argumentos, e tem como resultado 1 se pelo menos uma das entradas for igual a 1, e só é igual a 0 quando todas as entradas forem iguais a 0;
- Inversamente, a operação **AND** tem 0 como resultado se qualquer uma das entradas for igual a 0, e só é igual a 1 quando todas as entradas forem iguais a 1.

Expressões booleanas ou *expressões lógicas* são composições dessas operações básicas, geralmente escritas utilizando uma notação mais compacta:

- a' denota **NOT** a

[33] "George Boole - Wikipedia, the free encyclopedia", acessado 23 de fevereiro de 2012, http://en.wikipedia.org/wiki/George_Boole.
[34] "Claude Shannon - Wikipedia, the free encyclopedia".

- $a + b$ denota a **OR** b
- $a.b$ ou ab denota a **AND** b

Parênteses são usados em expressões booleanas da mesma forma que usamos em expressões aritméticas. Exemplos de expressões booleanas são $ab + a'b'$ e $(a + b)'$.

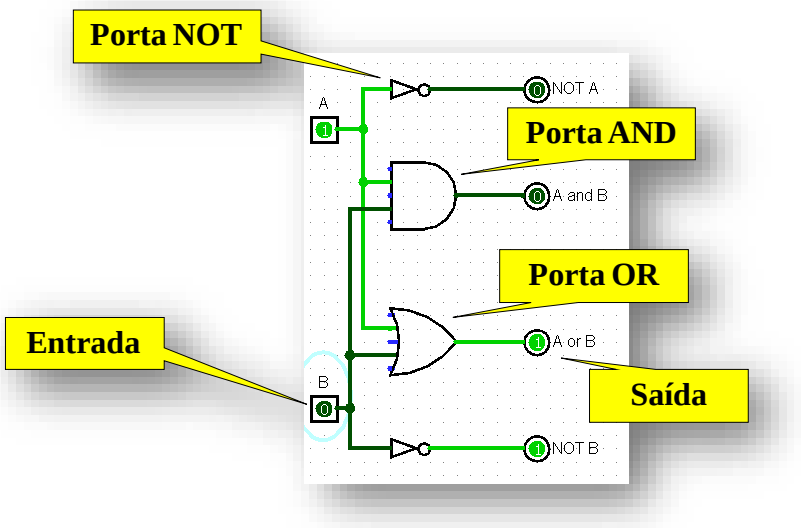


Figura 58: Portas lógicas, entradas e saídas em um diagrama de circuito

Circuitos digitais que implementam operações booleanas são conhecidos como *portas lógicas*. A Figura 58 mostra um diagrama de circuito contendo desenhos adotados por convenção para entradas, saídas e portas **NOT**, **AND** e **OR** e suas conexões.

Na álgebra de Boole diversas expressões são equivalentes; por exemplo,

$$(a')' = a; a + a = a; a.a = a$$

$$a + 1 = 1; a + 0 = a$$

$$a.1 = a; a.0 = 0$$

$$a + (b + c) = (a + b) + c = a + b + c$$

$$a.b + a.b' = a$$

Você pode demonstrar as propriedades acima simplesmente calculando as expressões para todos os valores possíveis para as variáveis envolvidas. Duas expressões equivalentes podem levar a circuitos equivalentes funcionalmente, mas com custos e desempenho distintos, como mostra a Figura 59. A simplificação de expressões booleanas é portanto uma área da mais alta importância, mas que foge do nosso escopo.

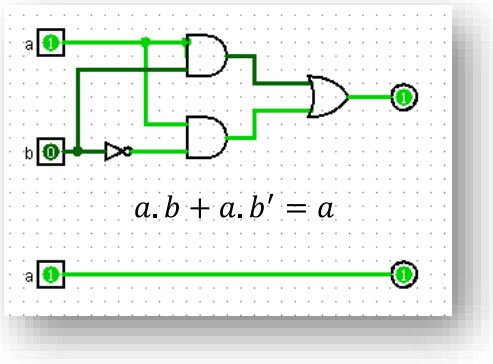


Figura 59: Dois circuitos funcionalmente equivalentes

Repare na convenção utilizada nos diagramas de circuito. Dois fios que se cruzam estão em contato elétrico somente se sua interseção for uma bolinha; sem esta bolinha, os fios se cruzam no desenho,mas não estão em contato.

2.3 Portas Lógicas e Transistores

Portas lógicas podem ser implantadas usando transistores, que são a base para a construção de circuitos compactos e rápidos. Por um lado, a tecnologia de 2011 permite colocar 2,6 bilhões de transistores em um chip de 1,6 mm de lado, e por outro, o tempo de reação de um desses transistores a mudanças em suas entradas é muito pequeno, da ordem de nano-segundos (10^{-9} segundos) ou mesmo de pico-segundos (10^{-12} segundos).

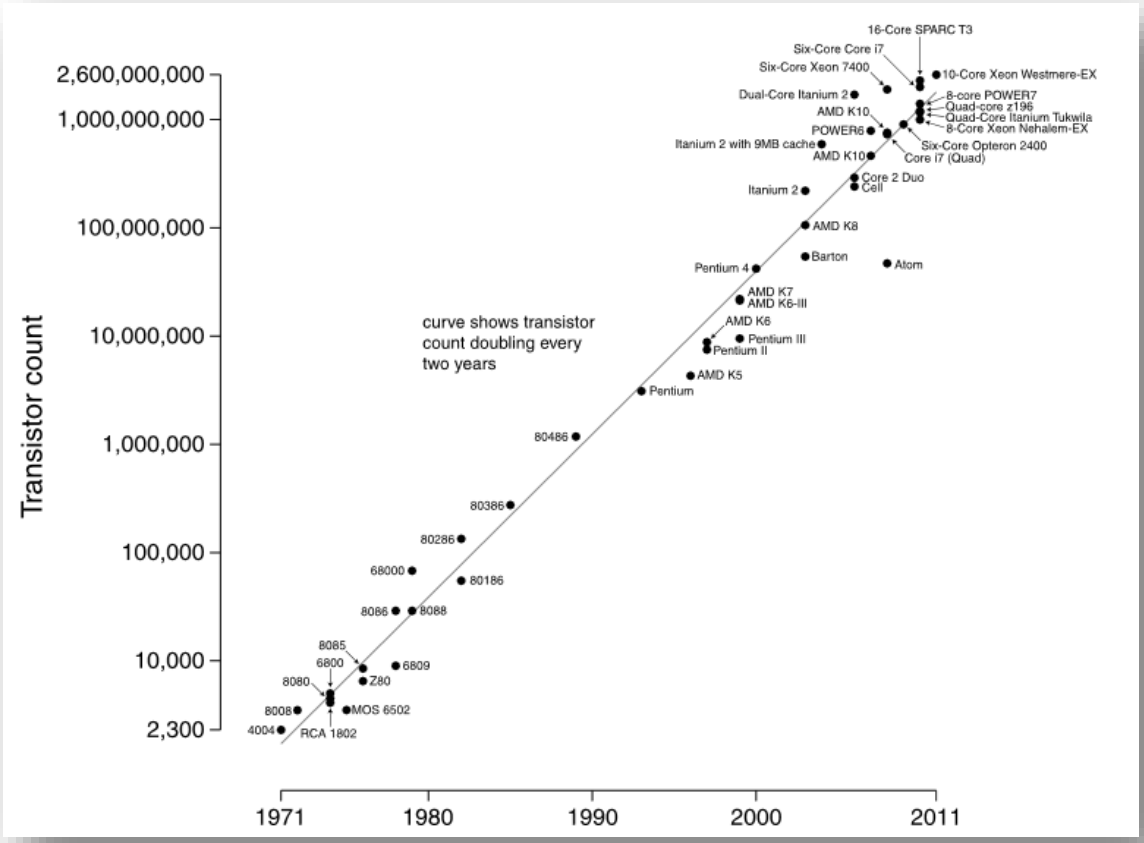


Figura 60: "Lei" de Moore: o número de transistores por área dobra a cada dois anos[35]

Duas operações booleanas que podem ser derivadas das operações básicas são **NAND** e **NOR**, cuja importância vem do fato de que sua construção com transistores é mais simples, como veremos a seguir. Estas operações são definidas por:

- a **NAND** $b = (a.b)'$
- a **NOR** $b = (a + b)'$

Ou seja, um **NAND** é um **AND** seguido de uma negação, assim como um **NOR** é um **OR** seguido de uma negação. A Figura 61 mostra os símbolos utilizados no desenho de circuitos lógicos para as portas **NAND** e **NOR**. Repare que o símbolo para **NAND** é quase igual ao símbolo usado para **AND**, diferenciando-se somente por um pequeno círculo em sua saída; o mesmo vale para o símbolo usado para **NOR**.

[35] "Moore's law - Wikipedia, the free encyclopedia", acessado 3 de março de 2013, http://en.wikipedia.org/wiki/Moore_Law.

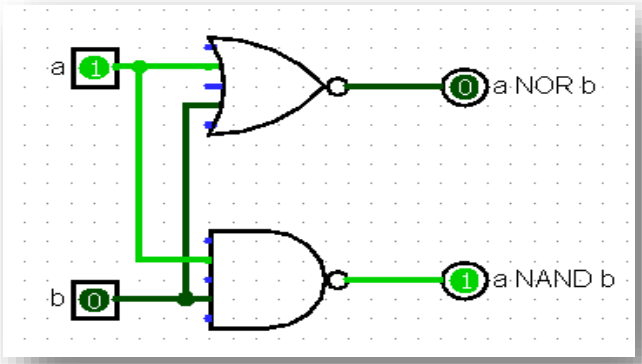


Figura 61: Símbolos para portas NAND e NOR

A Figura 62 mostra características que distinguem as operações AND, OR, NAND e NOR.

AND(a,b)	NAND(a,b)
Só é igual a 1 quando a e b são iguais a 1	Só é igual a 0 quando a e b são iguais a 1

OR(a,b)	NOR(a,b)
Só é igual a 0 quando a e b são iguais a 0	Só é igual a 1 quando a e b são iguais a 0

Figura 62: Características de AND, NAND, OR e NOR

Transistores podem ser e são utilizados para amplificar sinais, mas, em circuitos digitais, funcionam essencialmente como interruptores, trabalhando ora como condutores perfeitos, ora como isolantes perfeitos.

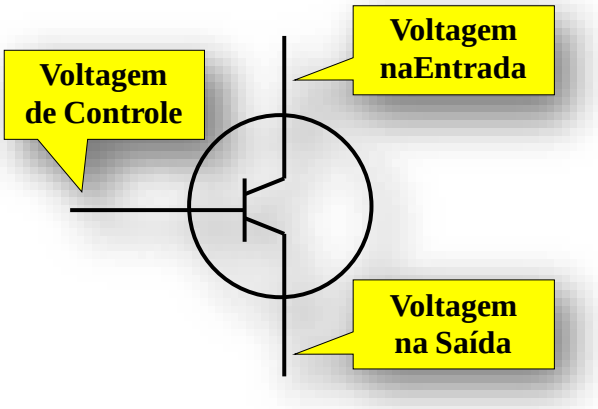


Figura 63: Símbolo de um transistor usado em diagramas de circuitos

Um transistor tem 3 pinos: um controle, uma entrada e uma saída. Em circuitos digitais o seu funcionamento se dá somente nas seguintes situações:

- quando a voltagem aplicada ao controle é alta (para algumas tecnologias, 5 volts), o transistor é um condutor perfeito, o que faz com que as voltagens na entrada e na saída sejam iguais;
- quando a voltagem aplicada ao controle é baixa (0 volts, digamos), o transistor é um isolante perfeito, e as voltagens na entrada e na saída podem diferir.

Vejamos inicialmente como uma porta **NOT** é implementada com a utilização de um transistor. Suponhamos que estamos representando o símbolo 0 por uma voltagem baixa e o símbolo 1 por uma voltagem alta. Como mostrado na Figura 64, uma fonte de voltagem alta é ligada à entrada do transistor, através de uma resistência, enquanto a saída do transistor é ligada à um ponto de terra. A "variável" que desejamos negar, a, é ligada ao controle do transistor. O resultado do circuito, a', é obtido no ponto entre a resistência e a entrada do transistor.

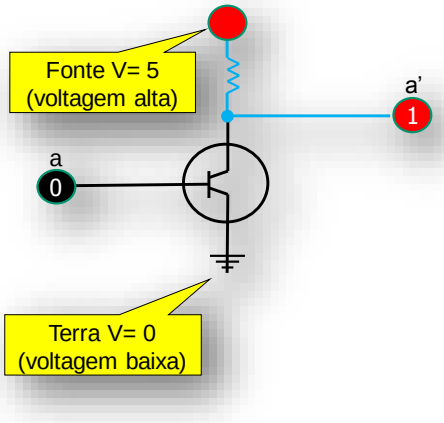


Figura 64: Uma porta NOT invertendo a = 0. Como a = 0, o transistor funciona como isolante perfeito

Quando $a = 0$, a voltagem aplicada ao controle do transistor é baixa e ele funciona como isolante perfeito, e obtemos $a' = 1$.

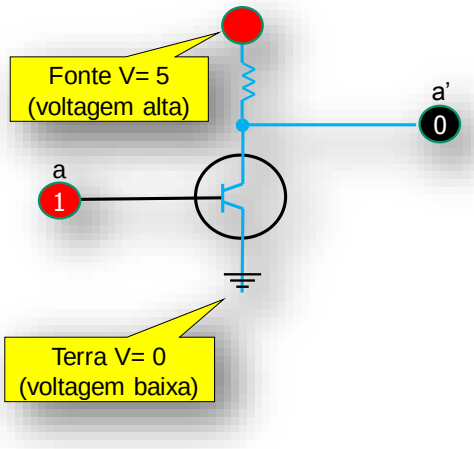


Figura 65: Porta NOT invertendo a = 1. O transistor funciona como condutor perfeito

Quando $a = 1$, o transistor funciona como condutor perfeito, e obtemos $a' = 0$, pois o contato com o ponto de terra é estabelecido.

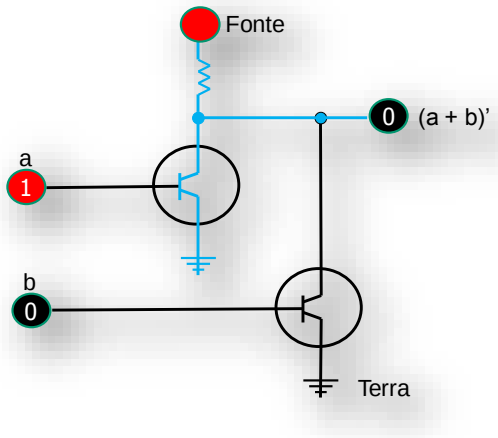


Figura 66: Uma porta NOR construída com dois transistores

Uma porta **NOR** é construída com o arranjo de transistores mostrado na Figura 66; não é difícil ver que o ponto $(a + b)'$ só terá o valor 1 (voltagem alta) quando os dois transistores do arranjo estiverem funcionando como isolantes, o que só ocorre quando $a = 0$ e $b = 0$.

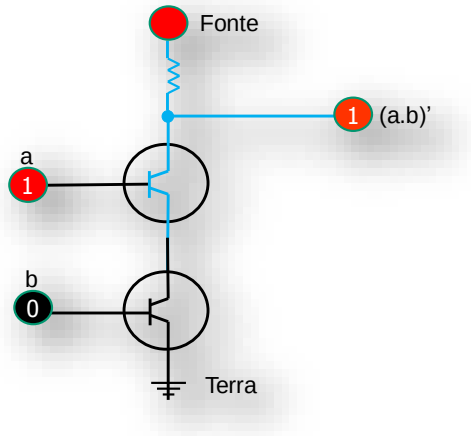


Figura 67: Uma porta NAND implantada com transistores

Uma porta **NAND** é construída de forma similar, mas com os transistores ligados em série, como mostra a Figura 67. Aqui a saída $(a.b)'$ só será igual a zero quando tivermos $a = 1$ e $b = 1$, valores que fazem com que a saída esteja conectada ao ponto de terra.

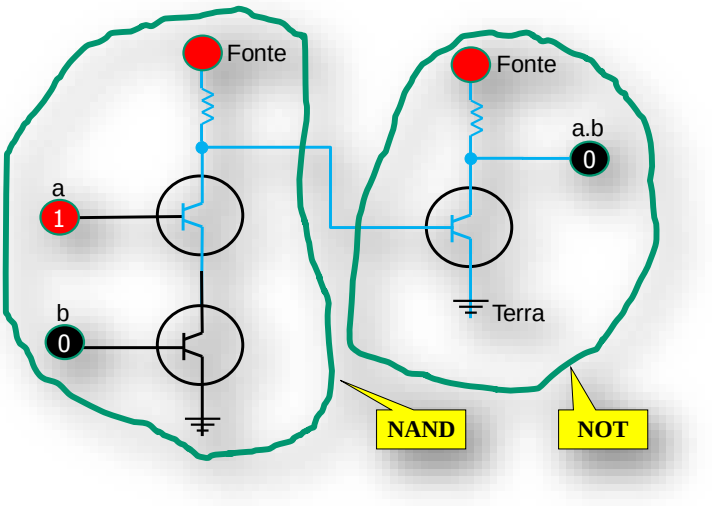


Figura 68: Porta AND com 3 transistores

Para obter uma porta **AND** usando transistores basta inverter a saída de uma porta NAND, como mostrado na Figura 68. Uma porta **OR** pode também ser obtida conectando a saída de uma porta **NOR** com a entrada de uma porta **NOT**.

2.4 Introdução ao Logisim

O Logisim[36] um simulador de circuitos lógicos, com objetivos didáticos, que você deve baixar pela Internet e instalar em seu computador. Este texto segue o tutorial que pode ser visto na ajuda do sistema.

Ao iniciar o Logisim você verá uma tela como a mostrada na Figura 69, talvez diferindo em alguns detalhes. Ali podemos ver uma barra de menu, ícons de componentes de uso frequente, um cardápio de componentes e um painel para o desenho efetivo de circuitos.

[36] Carl Burch, “Logisim”, acessado 20 de agosto de 2011, http://ozark.hendrix.edu/~burch/logisim/.

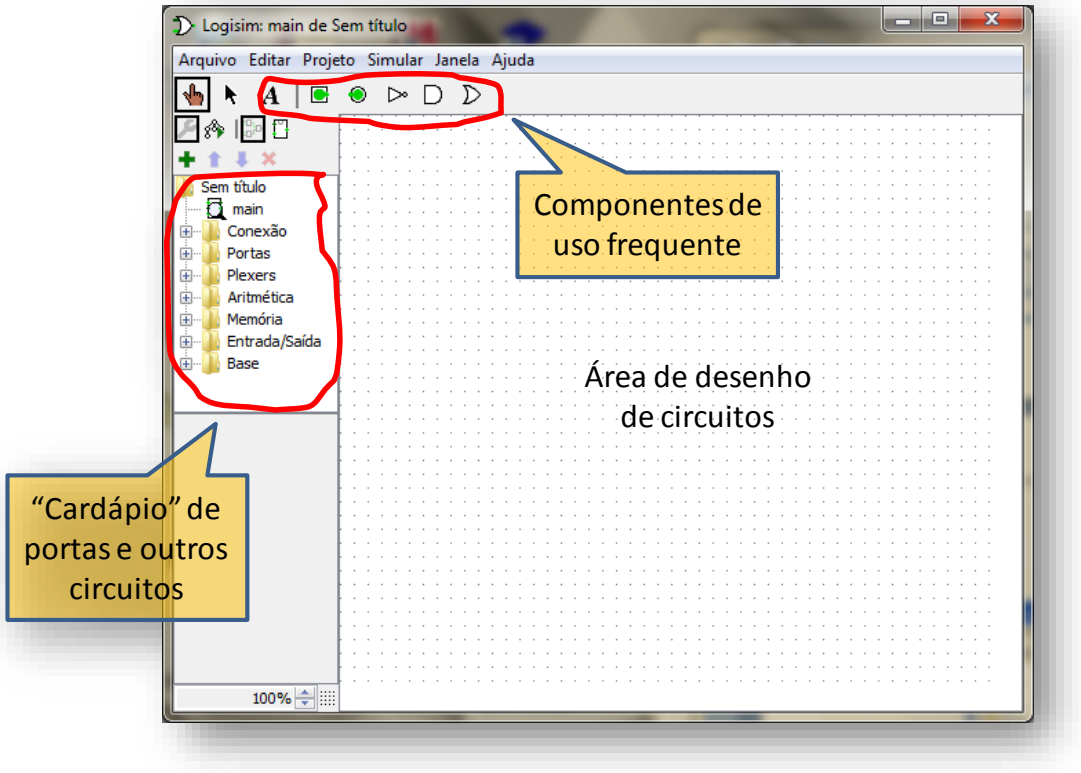


Figura 69: Tela inicial do Logisim

Para o nosso primeiro circuito, vamos implementar uma operação lógica importante, o **Ou Exclusivo**, escrito frequentemente como **XOR** (*exclusive or*). Se **a** e **b** são variáveis booleanas (isto é, variáveis cujos únicos valores possíveis são **Verdadeiro** e **Falso**, ou 0 e 1), **a XOR b** só tem o valor 1 (**Verdadeiro**) quando uma e somente uma das variáveis **a** e **b** tem o valor 1. O **Ou Exclusivo** não é uma operação primitiva da álgebra booleana, pois pode ser obtido através da expressão a **XOR b = ab' + a'b**.

Vamos começar colocando duas portas **AND**, clicando sobre o símbolo correspondente na barra de componentes de uso frequente, e posicionando as portas na área de desenho, como mostrado na Figura 70. Repare na tabela de atributos, que exibe e permite a edição de dados relativos ao elemento selecionado – no caso, a porta **AND** inferior.

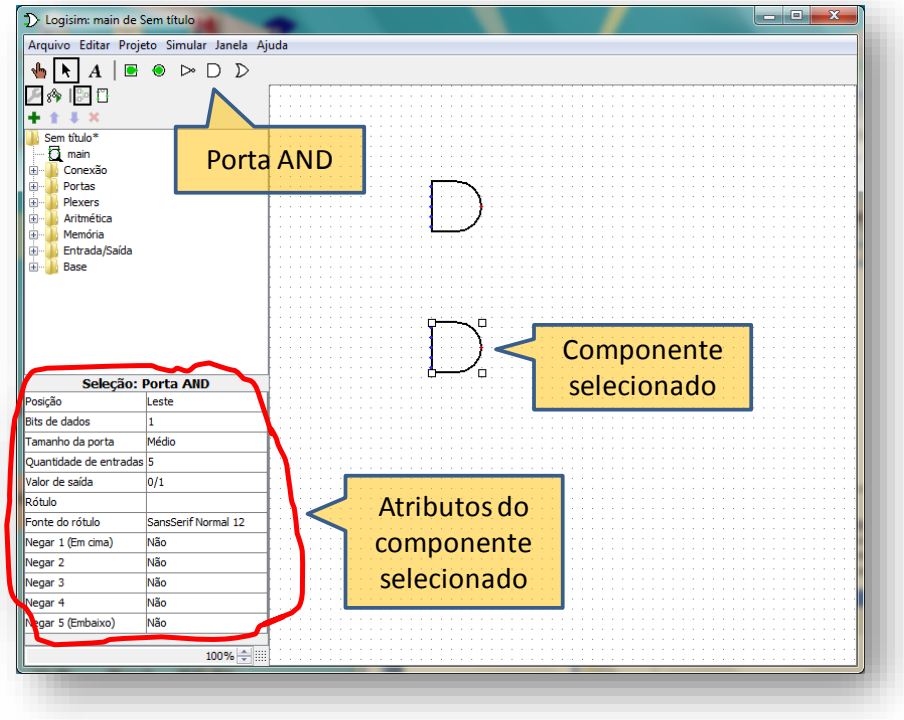


Figura 70: Duas portas AND

Depois, usando ainda a barra de ferramentas, vamos colocar uma porta **OR** e duas **NOT**, posicionando-as conforme a Figura 71.

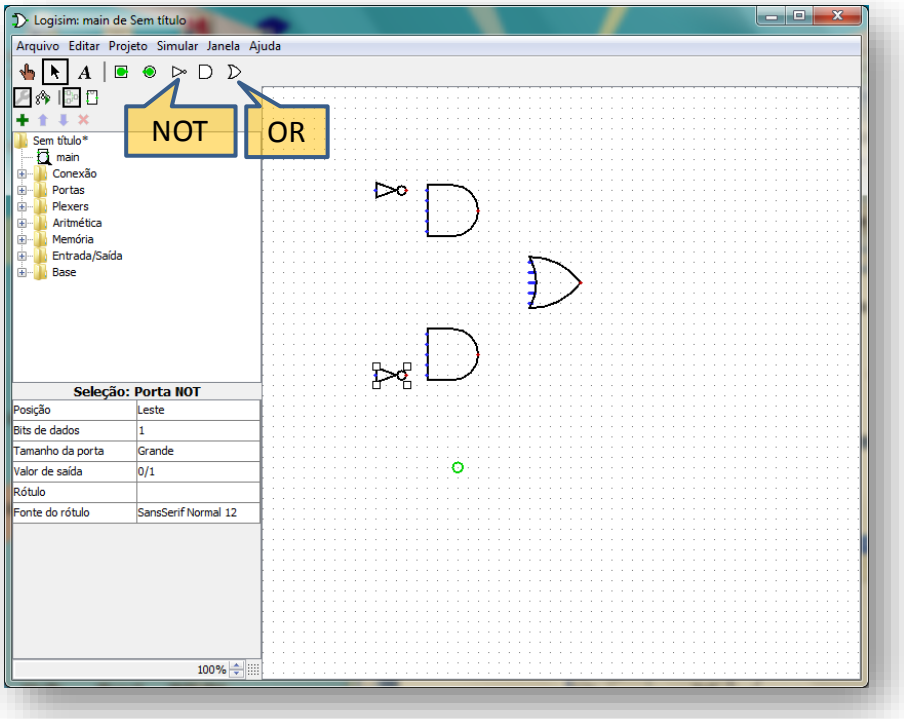


Figura 71: Acrescentando portas NOT e OR

O próximo passo é a colocação de entradas e saídas, que são colocadas usando os ícones em destaque na Figura 72.

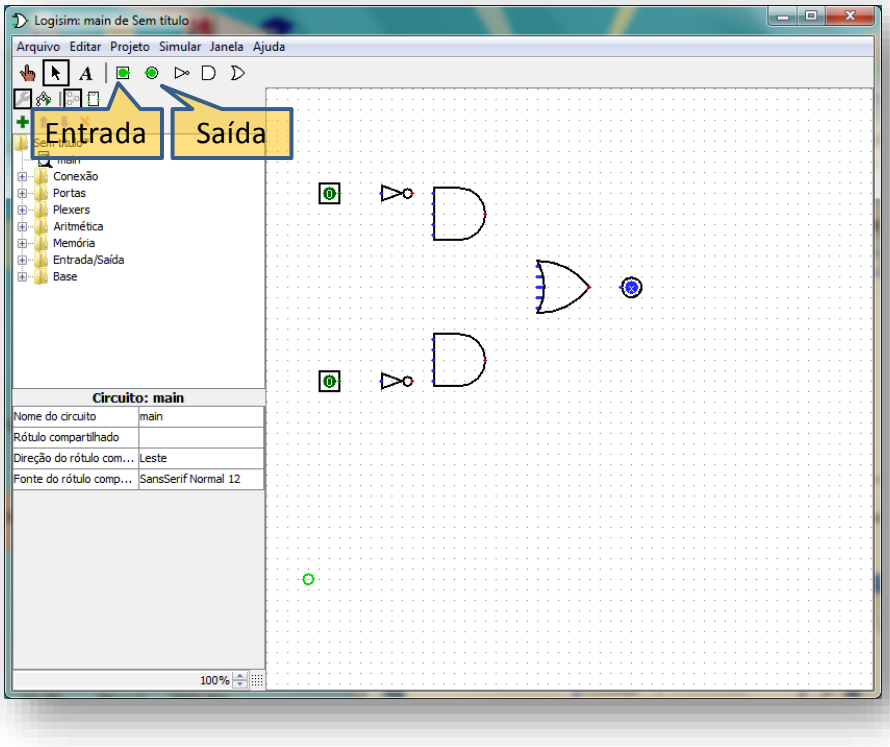


Figura 72: Acrescentando entradas e saídas

Para colocar “fios” ligando entradas, portas e saídas, utilize a ferramenta de seleção em destaque na Figura 73. Fios sempre seguem caminhos que são composições de segmentos horizontais ou verticais, chamados *caminhos Manhattan*.

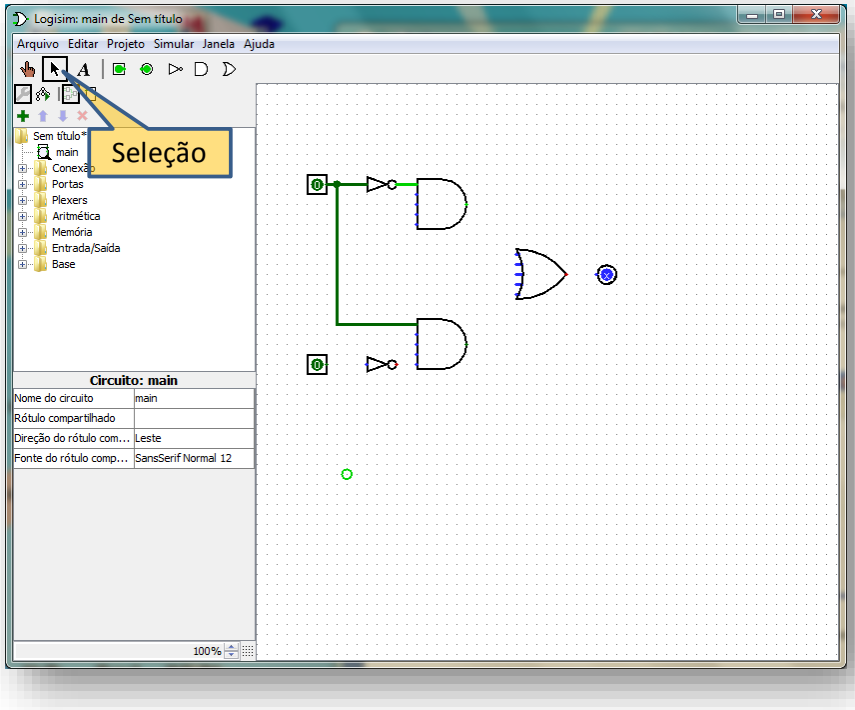


Figura 73: Acrescentando o cabeamento

Neste ponto você já pode notar convenções de cores usadas pelo Logisim: preto é usado para cabos com valor 0, verde para 1, e azul para cabos ainda sem valor determinado. Termine agora o cabeamento para obter o circuito da Figura 74 que implementa o **Ou Exclusivo**.

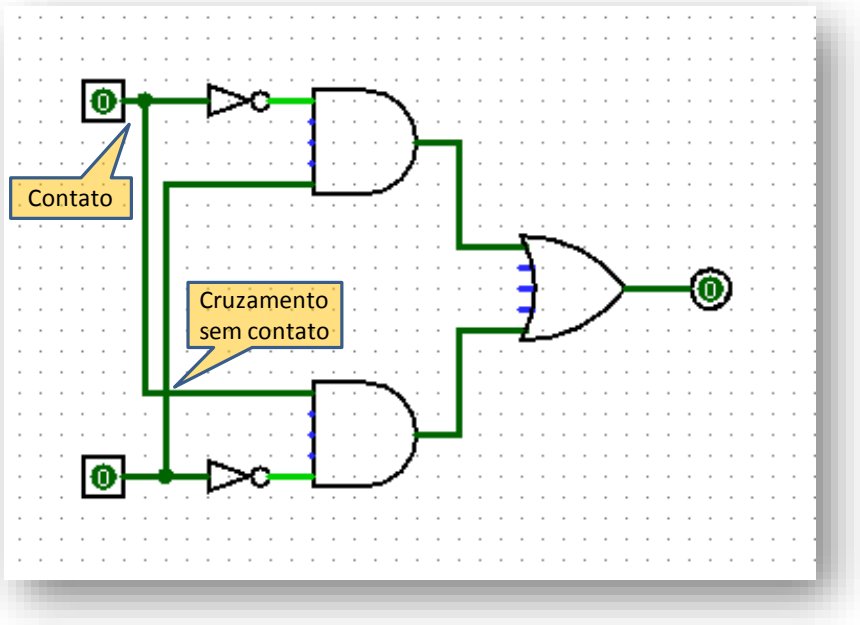


Figura 74: Circuito OU Exclusivo

Seu diagrama ficará mais fácil de ser compreendido se você acrescentar textos, usando a ferramenta A em destaque na Figura 75. Você pode alterar as características da fonte (tamanho, negrito ou itálico, etc.) editando os atributos do texto selecionado.

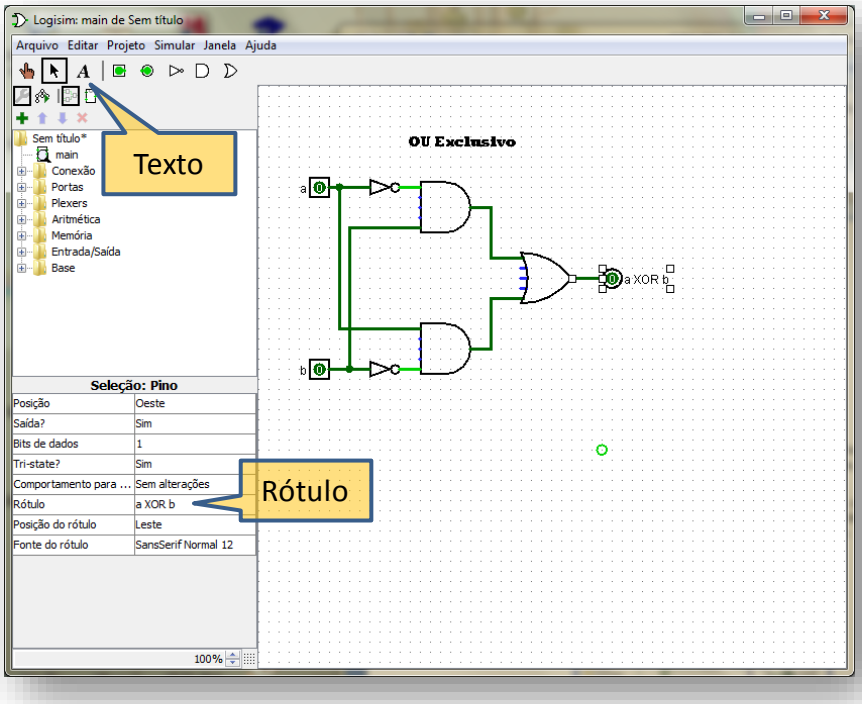


Figura 75: Acrescentando textos a um circuito

Textos podem ser colocados em qualquer posição na área de desenho, mas muitas vezes é melhor colocá-los como "rótulos" de elementos de circuito, como entradas, portas lógicas e saídas. O rótulo acompanha o elemento de circuito quando este é movido para outra posição. Para isto, selecione o elemento, e preencha o campo *Rótulo* na tabela de atributos, como mostrado na Figura 75.

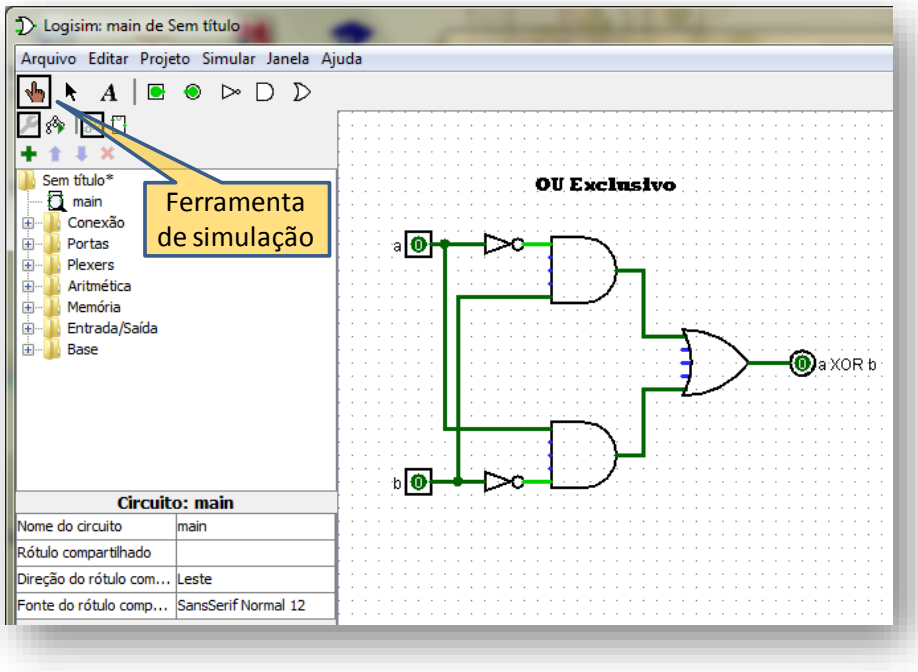


Figura 76: Ferramenta de simulação

Para testar o seu circuito, use a ferramenta de simulação – a mãozinha em destaque na Figura 76. Utilize-a clicando sobre as entradas do circuito para alterar o seu valor. Explore todas as combinações possíveis de valores para a e b, verificando a saída para completar a tabela da Figura 77.

A	B	a XOR b
0	0	
0	1	
1	0	
1	1	

Figura 77: Complete com os valores produzidos pelo seu circuito XOR

Circuitos podem ser salvos em arquivos para uso posterior. Para salvar o seu circuito, use a opção *Arquivo/Salvar* do menu do Logisim, escolha um nome para o arquivo e um diretório, e salve-o. Para voltar a trabalhar com o arquivo, use *Arquivo/Abrir*.

Para introduzir elementos de circuito com outras orientações, clique sobre a porta desejada, e depois altere o campo *Posição* na tabela de atributos, conforme mostrado na Figura 78.

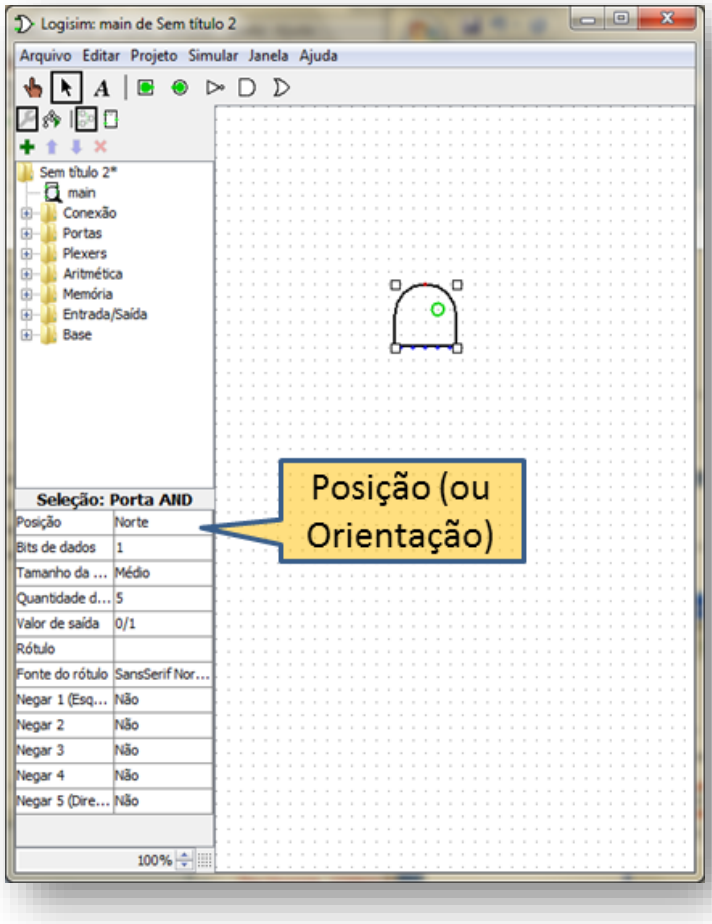


Figura 78: Mudando a orientação de um componente de circuito

Se você seguiu todos os passos deste roteiro, terá assimilado os elementos básicos para o uso do Logisim. Você pode descobrir muito mais lendo a ajuda ou explorando diretamente a ferramenta.

2.5 Soma de duas parcelas de 1 bit

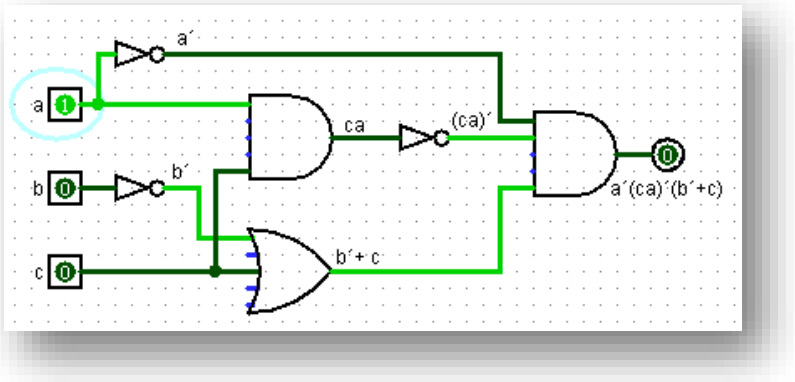


Figura 79: Circuito correspondente à expressão a'(ca)'(b'+c)

Nós vimos como circuitos lógicos implementam expressões lógicas, com um mapeamento direto. Por exemplo, a Figura 79 mostra um circuito que corresponde à expressão $a'(ca)'(b' + c)$. Não é difícil acreditar que conseguimos obter circuitos para qualquer expressão lógica. Mas podemos fazer circuitos lógicos que fazem contas? Ora, como o nome indica, computadores

computam, isto é, fazem contas, e sabemos portanto que a resposta é afirmativa. Mas como? É o que veremos nesta seção.

Vamos começar por um problema bem simples: encontrar um circuito lógico que faça a soma de dois inteiros representados como binários sem sinal de 1 bit cada um. Na base dez, o resultado pode ser 0, 1 ou 2. Portanto o resultado, também codificado como binário sem sinal, pode ser 00, 01 ou 10. Ou seja, serão necessários 2 bits para representar o resultado da soma de duas variáveis de 1 bit.

É importante entender que *somar* significa obter a representação como um binário sem sinal da soma das entradas , entendidas também como binários sem sinal. Na verdade, calcular ou computar uma função qualquer significa transformar a representação de seus argumentos na representação do valor da função, coisa que já fazemos ao seguir as regras da aritmética que aprendemos no ensino primário. A Figura 80 mostra as entradas e saídas do circuito que pretendemos construir.

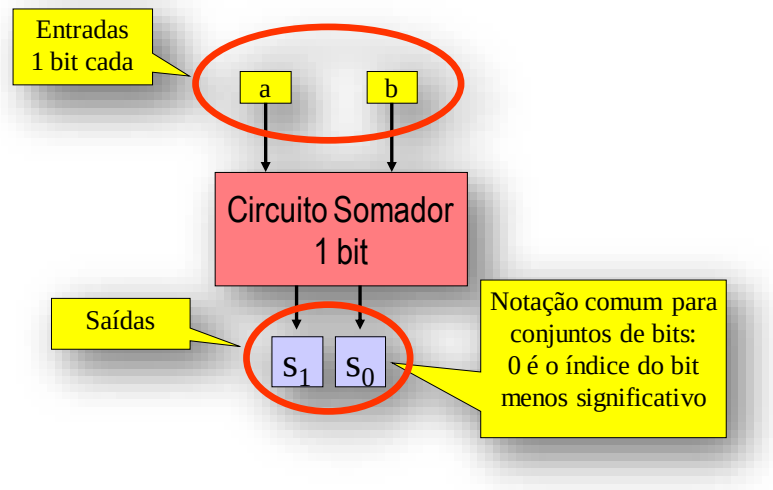


Figura 80: Entradas e saídas de um somador de 1 bit

Para especificar exatamente qual a transformação de informação que esperamos deste circuito, vamos utilizar uma *tabela da verdade*. Uma tabela da verdade apresenta o valor de uma ou mais funções lógicas – as saídas do circuito – correspondendo a cada combinação possível de valores das variáveis de entrada. A Figura 81 mostra a tabela da verdade para a soma de duas variáveis de 1 bit.

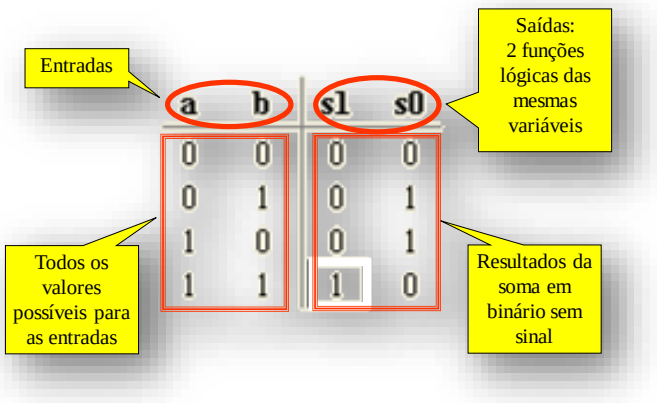

Entradas
Saídas:
2 funções lógicas das mesmas variáveis
a b s1 s0
0 0 0 0
0 1 0 1
1 0 0 1
1 1 1 0
Todos os valores possíveis para as entradas
Resultados da soma em binário sem sinal

Figura 81: Tabela da Verdade para soma de duas variáveis de 1 bit

Nessa tabela nós temos:

- duas variáveis de entrada, a e b;
- duas funções de saída, s_1 e s_0; cada função corresponde a um dígito binário do resultado;
- a cada linha da tabela corresponde uma combinação das variáveis de entrada e o valor correspondente das funções de saída. Como são duas variáveis, temos $2^2 = 4$ linhas na tabela.

Tabelas da verdade constituem um mecanismo geral para a especificação de funções lógicas. Elas especificam as saídas para cada combinação possível das variáveis de entrada. Nós já vimos tabelas da verdade quando introduzimos as funções **NOT**, **AND** e **OR**, na Seção 2.2.

Dada uma expressão lógica, nós podemos construir sua tabela da verdade efetuando as operações da expressão. A Figura 82 mostra a tabela da verdade para a expressão $a'(b + c)$, contendo valores intermediários usados no cálculo do valor da expressão.

a	b	c	a'	$b + c$	$a'(b + c)$
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	1	1	1
0	1	1	1	1	1
1	0	0	0	0	0
1	0	1	0	1	0
1	1	0	0	1	0
1	1	1	0	1	0

Figura 82: Tabela da verdade para a expressão $a'(b + c)$

Mas nosso problema não é bem esse. Nós temos uma tabela da verdade e, para obter um circuito lógico para uma das funções de saída a partir de uma tabela da verdade é preciso:

- conseguir uma expressão lógica equivalente à tabela da verdade, e
- construir o circuito equivalente à expressão lógica.

Uma expressão lógica e uma função definida por uma tabela da verdade são equivalentes quando para qualquer combinação de valores das variáveis de entrada, os valores da função e os valores da expressão são iguais. Por exemplo, a função $\mathrm{f(x, y)}$ definida pela tabela mostrada na Figura 83 é equivalente à expressão $f(x, y) = x'y' + xy$.

x	y	f(x,y)
0	0	1
0	1	0
1	0	0
1	1	1

Figura 83: Tabela da verdade para a função f(x,y)

Um método genérico para se obter uma expressão lógica para uma função definida por uma tabela da verdade consiste em fazer um **OR** de termos que só têm o valor 1 para cada combinação das variáveis de entrada para a qual o valor da função é igual a 1.

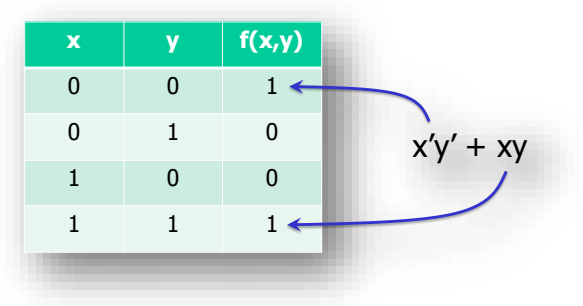


Figura 84: Cobertura dos 1s de uma função booleana

Na Figura 84, $f(x,y) = 1$ somente na primeira linha, quando $x = 0$ e $y = 0$, e na última linha, quando $x = 1$ e $y = 1$. Nós temos:

- $x'y' = 1$ se e somente se $x = 0$ e $y = 0$; para qualquer outra combinação de valores de x e y, $x'y' = 0$
- $xy = 1$ se e somente se $x = 1$ e $y = 1$.; para qualquer outra combinação de valores de x e y, $xy = 0$.

Portanto, o **OR** destes dois termos, $f(x,y) = x'y' + xy$, "cobre" exatamente os 1s da tabela, e é uma expressão lógica equivalente à função desejada. A expressão assim obtida é chamada de *soma canônica* associada à função.

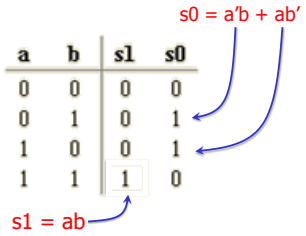


Figura 85: Expressões lógicas para um somador de duas variáveis de 1 bit

Retornando ao problema do somador de 1 bit, temos duas funções para as quais queremos encontrar expressões lógicas equivalentes. Na Figura 85 vemos que, para a função s_1, temos apenas o 1 da última linha a cobrir, o que é feito pelo termo ab, e para a função s_0, temos um 1 na segunda linha, coberto pelo termo $a'b$, e outro 1 na terceira linha, coberto pelo termo ab'. Ou seja, as expressões lógicas que desejamos são $s_0 = a'b + ab'$ e $s_1 = ab$. A Figura 86 mostra um circuito que implementa essas expressões lógicas. Este circuito é conhecido como "circuito meia-soma", por contraste com o circuito "soma-completa" que veremos a seguir.

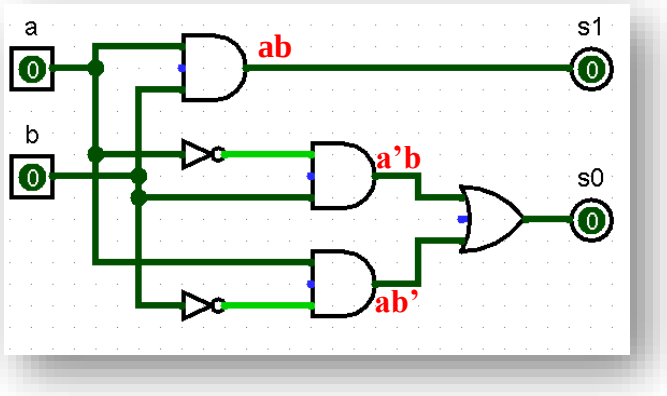


Figura 86: Circuito lógico para soma binária de 1 bit, ou circuito meia-soma

Temos aqui um resultado importantíssimo: construimos um circuito lógico com duas entradas de 1 bit representando binários sem sinal, e que produz em sua saída dois bits representando a soma das entradas, também codificado como um binário sem sinal. Como prometemos, conseguimos fazer contas com as funções lógicas NOT, AND e OR.

2.6 Soma de dois inteiros com vários bits

Tudo bem, conseguimos fazer um circuito que realiza a soma de duas variáveis de 1 bit. Uma façanha, mas em termos práticos, muito pouco. Para fazer qualquer coisa mais séria, precisamos ser capazes de somar variáveis de, digamos, 32 bits, como são representados inteiros em boa parte dos computadores atuais.

Em princípio nós poderíamos construir uma tabela da verdade com as entradas e saídas, e depois inferir expressões lógicas para as saídas usando o método de cobertura dos 1s. Essa tabela teria 64 bits de entrada, correspondendo às duas variáveis a serem somadas, e 33 bits de saída, pois a soma de duas variáveis de 32 bits pode ter 33 bits. O problema é que essa tabela teria 2^{64} ou aproximadamente $10^{64/3,3} \cong 10^{19}$ linhas. Para entender, considere a equação

$$2^n = 10^x = \left(2^{log_2(10)}\right)^x = 2^{xlog_2(10)},$$

ou

$$x = \frac{n}{log_2(10)} \cong \frac{n}{3,3}).$$

Vamos nos situar. Um ano tem somente $3,1536 \times 10^{16}$ nano-segundos, e um nano-segundo é a ordem de grandeza do tempo de reação de portas lógicas na tecnologia atual. Se considerarmos que a análise de cada linha da tabela tomaria 1 nano-segundo, o que é de um enorme otimismo, a obtenção da expressão para a soma tomaria no mínimo 3.000 anos. Ou seja, mesmo se teoricamente conseguiríamos construir o somador de 32 bits pelo método de cobertura dos 1s, na prática isso é absolutamente inviável.

Temos portanto que adotar outro enfoque. Primeiramente vamos examinar como nós, humanos, fazemos uma soma em binário. Depois, vamos construir um somador de forma modular, explorando regularidades no processo de soma.

Números binários adotam a notação posicional, a mesma adotada no sistema decimal que usamos no dia a dia, cujo imenso sucesso se deve exatamente à facilidade com que operações aritméticas são feitas. Uma soma de números binários é similar à soma decimal que aprendemos na escola primária – um algoritmo venerável, difundido no Ocidente por ninguém

menos que Muḥammad ibn Mūsā al-Khwārizmī, matemático árabe[37] que viveu entre 780 e 850 em Bagdá, e cujo nome traduzido para o latim deu origem à palavra *algoritmo*. Para entender o sucesso do então novo método, que concorria com números romanos, responda rábido, quanto vale LIII + XXVIII? E, pior, quanto vale LIII × XXVIII?

Para fazer uma soma na base 10 nós somamos algarismo por algarismo, da direita (menos significativos) para a esquerda (mais significativos). Quando a soma de dois algarismos excede 9, colocamos como resultado daquela coluna somente o seu dígito mais significativo, e acrescentamos 1 (o "vai-um") na soma da coluna seguinte (Figura 87).

Vai-um	1	1		
a	2	7	7	6
b	4	7	9	1
a+b	7	5	6	7

Figura 87: Soma em decimal

Para somar dois números binários o procedimento é análogo. Somamos bit a bit, da direita para a esquerda e, quando a soma de uma coluna excede 1, colocamos como resultado da coluna somente o bit mais significativo de sua soma, e temos um "vai-um" para a coluna seguinte. A Figura 88 mostra o processo do cálculo de 1011101+1001110.

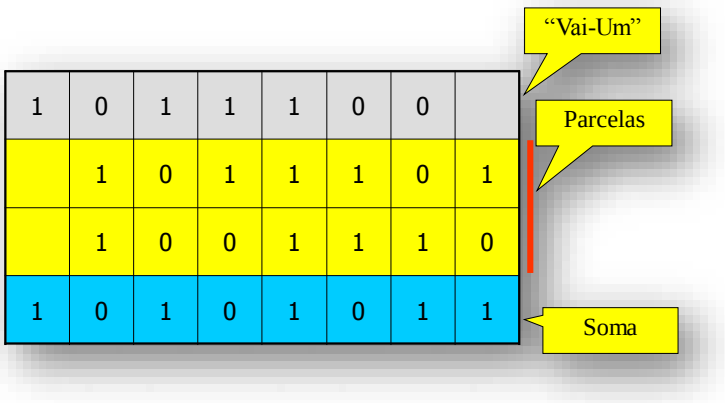


Figura 88: Exemplo de soma de binários

Nós vamos usar essa aritmética para construir um circuito somador de vários bits. A idéia é construir um módulo que faça a operação de uma das colunas da soma. O somador terá tantos deste módulo quantas forem as colunas, ou seja, tantos módulos quantos forem os bits das parcelas.

[37] "Muḥammad ibn Mūsā al-Khwārizmī - Wikipedia, the free encyclopedia".

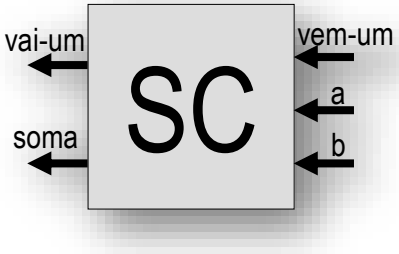


Figura 89: Entradas e saídas de um circuito de soma completa

Este módulo, esquematizado na Figura 89, terá três entradas:

- os dois bits de uma coluna das parcelas, e
- uma entrada que vamos chamar de "vem-um", que vamos ligar à saída "vai-um" da coluna à direita,

As saídas do módulo são duas:

- um bit de resultado
- um bit de "vai-um", que será ligado à entrada "vem-um" da coluna à esquerda.

Um circuito que implemente esse módulo é conhecido como "circuito soma-completa".

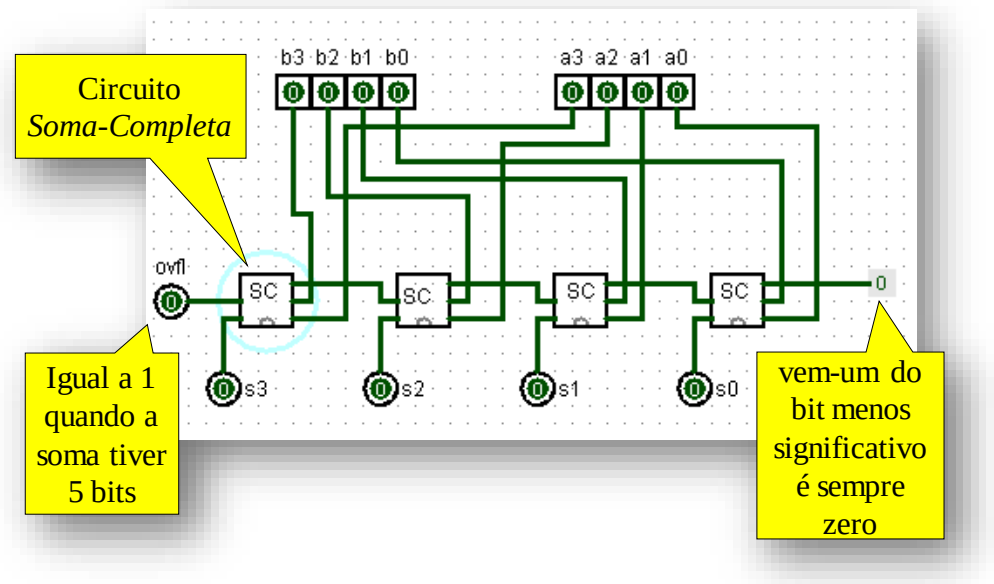


Figura 90: Arranjo em cascata de módulos para um somador de 4 bits

A Figura 90 mostra um arranjo em cascata de circuitos de soma completa, capaz de somar duas variáveis de 4 bits. O módulo SC (Soma Completa) mais à direita faz a soma dos bits menos significativos de cada parcela, a_0 e b_0. Sua entrada "vem-um" recebe o valor constante 0. Ele produz um bit de resultado, s_0, e o vai-um da coluna, que é conectado à entrada vem-um do segundo SC da direita para a esquerda. Este segundo SC faz a soma de a_1, b_1 e de sua entrada vem-um que, como dissemos, é o vai-um do primeiro SC. E esse arranjo se repete, aqui para os 4 bits das parcelas, mas poderia se repetir por 32, ou por 64, ou por 128 vezes. Em um somador de n bits, o circuito de soma completa do bit mais significativo produz um vai-um que indica que a soma das parcelas só pode ser representada com $n + 1$ bits. Essa situação é designada por "estouro", ou mais comumente, pelo termo em inglês *overflow*.

soma = a'b'v + a'bv' + ab'v' + abv

a	b	vem-um	soma	vai-um
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

vai-um = a'bv + ab'v + abv' + abv

Figura 91: Tabela da verdade para um circuito de soma completa

A Figura 91 mostra a tabela da verdade para um circuito de soma completa, e as expressões encontradas pelo método de cobertura dos 1s para as saídas soma e vai-um. A variável v designa aqui a entrada vem-um.

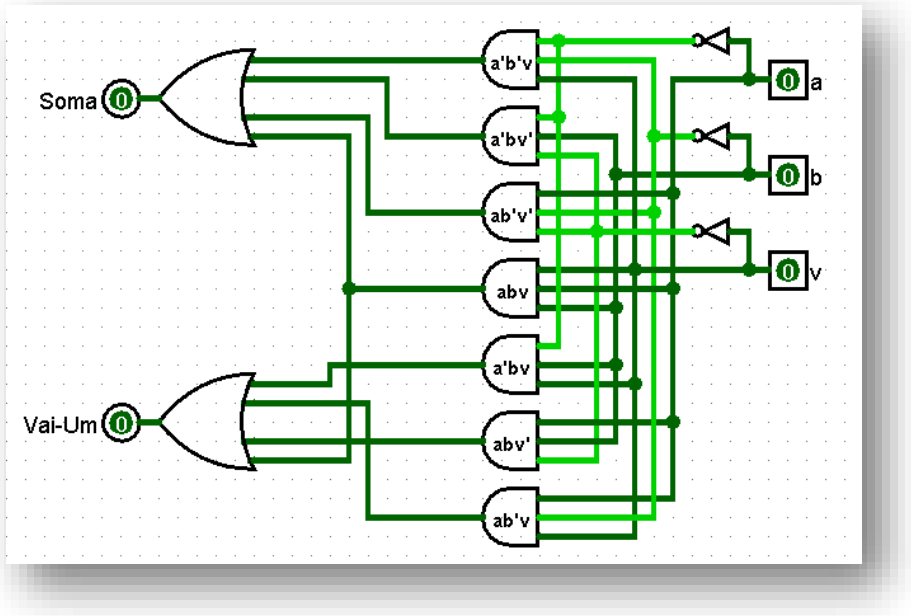


Figura 92: Circuito Soma Completa, construído diretamente a partir das expressões obtidas da tabela da verdade

A Figura 92 mostra um circuito de soma completa construído com a utilização destas expressões lógicas.

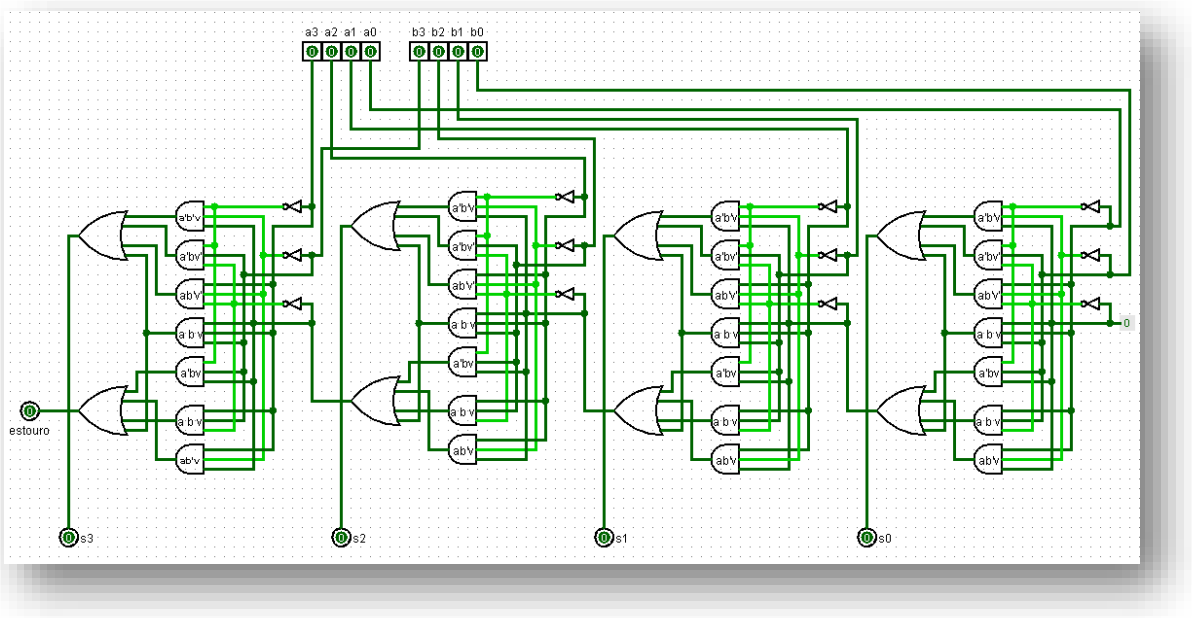


Figura 93 Somador de 4 bits, usando 4 circuitos de Soma Completa

Podemos agora obter somadores de n bits, compondo n circuitos de Soma Completa seguindo o arranjo proposto na Figura 90. A Figura 93 mostra um somador de 4 bits obtido com este desenho.

2.7 Modularização e Síntese Automática de Circuitos Combinatórios no Logisim

A extração de uma expressão lógica (e do circuito correspondente) a partir de uma tabela da verdade é um processo que pode ser automatizado, e o Logisim oferece esta possibilidade. Vamos aqui reconstruir o circuito de soma completa a partir de sua tabela da verdade, usando o Logisim, e vamos aproveitar para aprender como construir e utilizar módulos neste sistema.

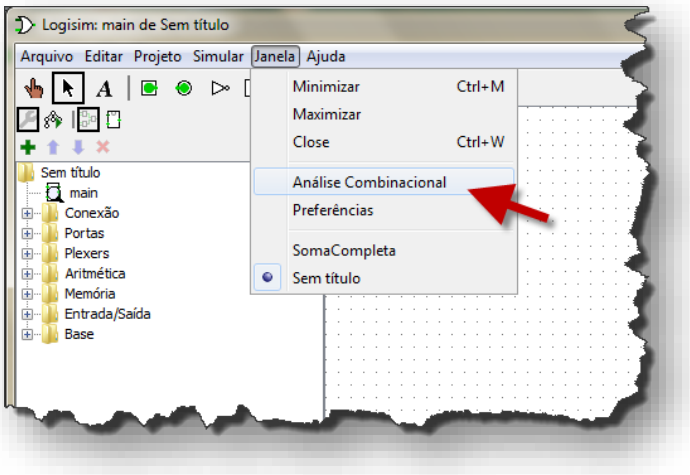


Figura 94: Abrindo a janela de análise combinacional do Logisim

Abra o Logisim, e siga o menu *Janela/Análise Combinacional*, como mostrado na Figura 94. Na janela *Análise Combinacional*, na aba *Entradas*, insira os nomes `a`, `b` e `v`, que correspondem às entradas de um circuito de soma completa, como mostra a Figura 95.

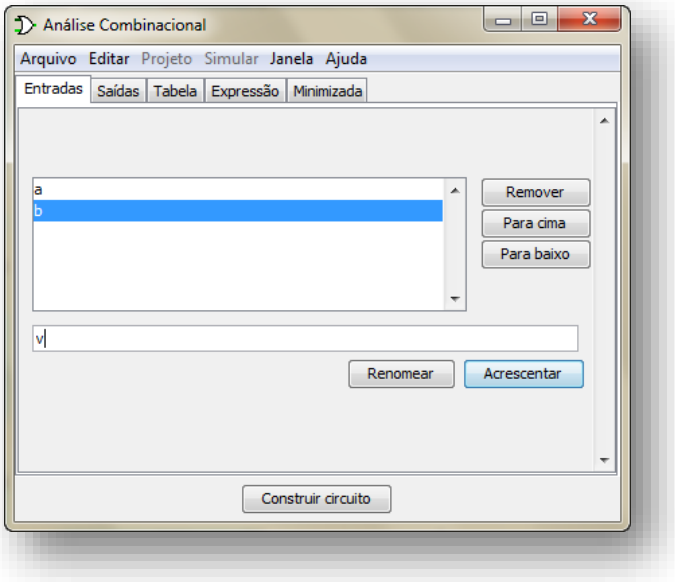


Figura 95: Definindo no Logisim as entradas do circuito

Na aba *Saídas*, insira os nomes `VaiUm` e `Soma`, conforme mostra a Figura 96.

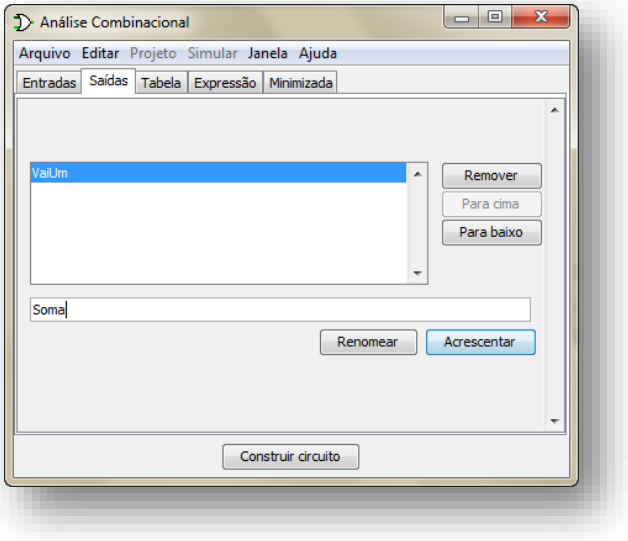


Figura 96: Definindo no Logisim as saídas de um circuito

Depois, clique sobre a aba *Tabela*, e entre com a tabela da verdade para o circuito de soma completa, com os valores mostrados na Figura 91.

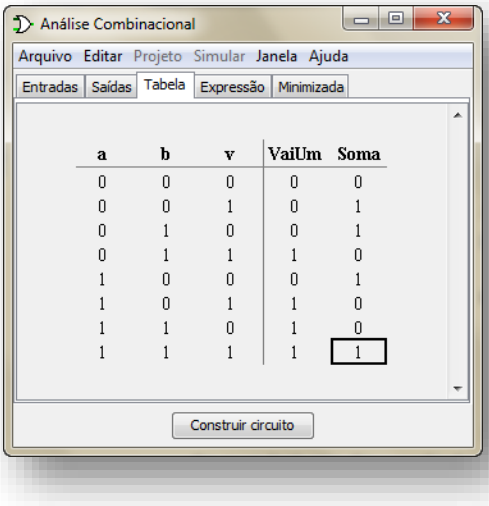

Análise Combinacional
Arquivo Editar Projeto Simular Janela Ajuda
Entradas Saídas Tabela Expressão Minimizada
a b v VaiUm Soma
0 0 0 0 0
0 0 1 0 1
0 1 0 0 1
0 1 1 1 0
1 0 0 0 1
1 0 1 1 0
1 1 0 1 0
1 1 1 1 1
Construir circuito

Figura 97: Entrando com a tabela da verdade para a soma completa no Logisim

A Figura 97 mostra a tabela já com todos os valores inseridos. Isso é feito clicando uma ou mais vezes sobre cada ponto nas colunas de saída da tabela; a cada clique, o valor é trocado, indo de `x` para `0`, de `0` para `1` ou de `1` para `x`.

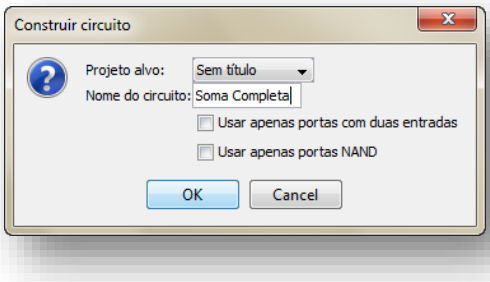


Figura 98: Escolha do nome do novo circuito no Logisim

Quando a tabela estiver pronta, clique sobre o botão *Construir Circuito*, e dê ao novo circuito o nome de `Soma Completa`, como mostrado na Figura 98.

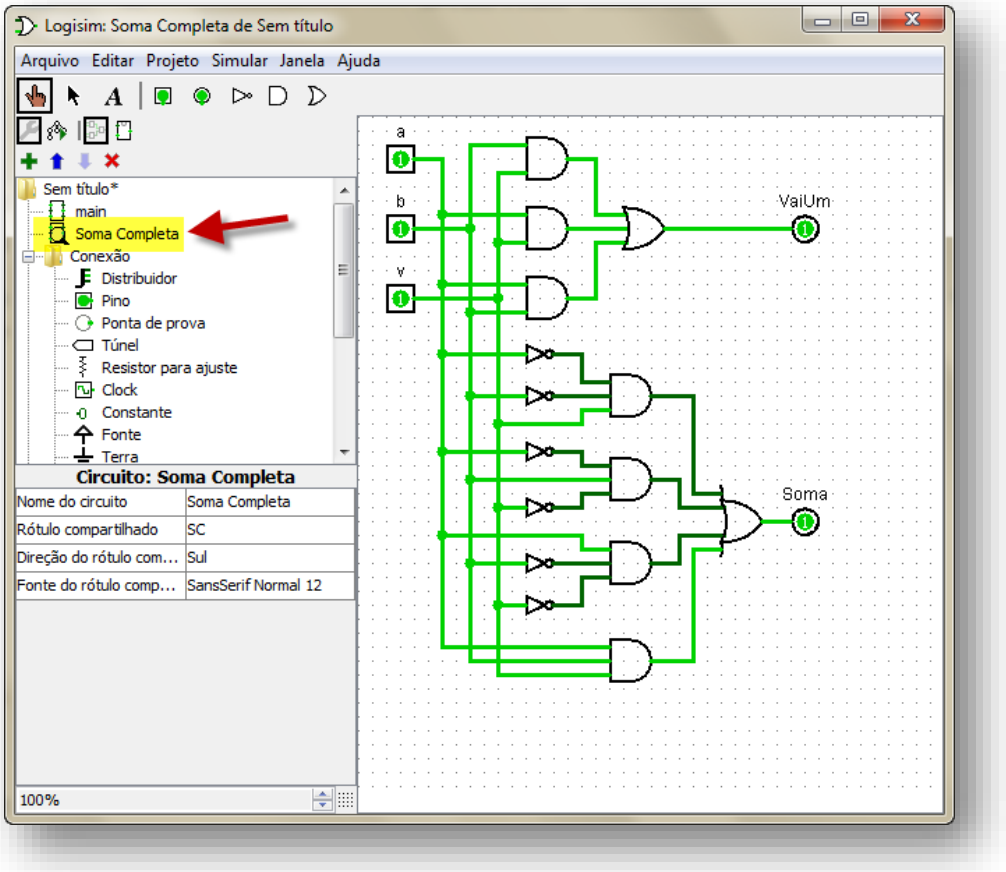


Figura 99: Circuito de soma completa gerado automaticamente pelo Logisim a partir da tabela da verdade

Aperte o botão *OK* e... como mostra a Figura 99, nosso circuito de soma completa está pronto! Mas está um pouco diferente do circuito que já construímos, mostrado na Figura 92. No circuito sintetizado automaticamente pelo Logisim a saída vai-um é dada pela expressão

$$V = bv + av + ab$$

onde V designa a saída Vai-um. Isto não é um erro, pois esta expressão é equivalente à expressão $V = a'bv + ab'v + abv' + abv$ encontrada pelo método de cobertura de 1s, o que você pode verificar construindo uma tabela da verdade para as duas expressões. Pelo contrário, a simplificação da expressão leva a um circuito mais simples e funcionalmente equivalente. Percebemos que o Logisim não somente realiza automaticamente o nosso método de cobertura dos 1s, como também é capaz de simplificar a expressão resultante.

Vamos agora refazer o somador de 4 bits, desta vez usando módulos. É importante você reparar que o Logisim criou o circuito *Soma Completa* como um sub-circuito, o que é indicado no painel de componentes – veja a seta na Figura 99. Para isto dê um duplo clique sobre o sub-circuito *main*. Você verá o circuito *main*, vazio. Agora, dê um clique sobre o sub-circuito *Soma Completa* no painel de componentes, dirija o mouse para o painel de desenho, e dê outro clique na posição onde você deseja colocar o primeiro sub-circuito de *Soma Completa*. Este sub-circuito aparece agora como um bloquinho sem detalhes internos, com pequenos pontos marcando seus pinos de entrada e saída. Passe com o mouse sobre essas entradas e saídas; o Logisim indica o nome de cada uma delas.

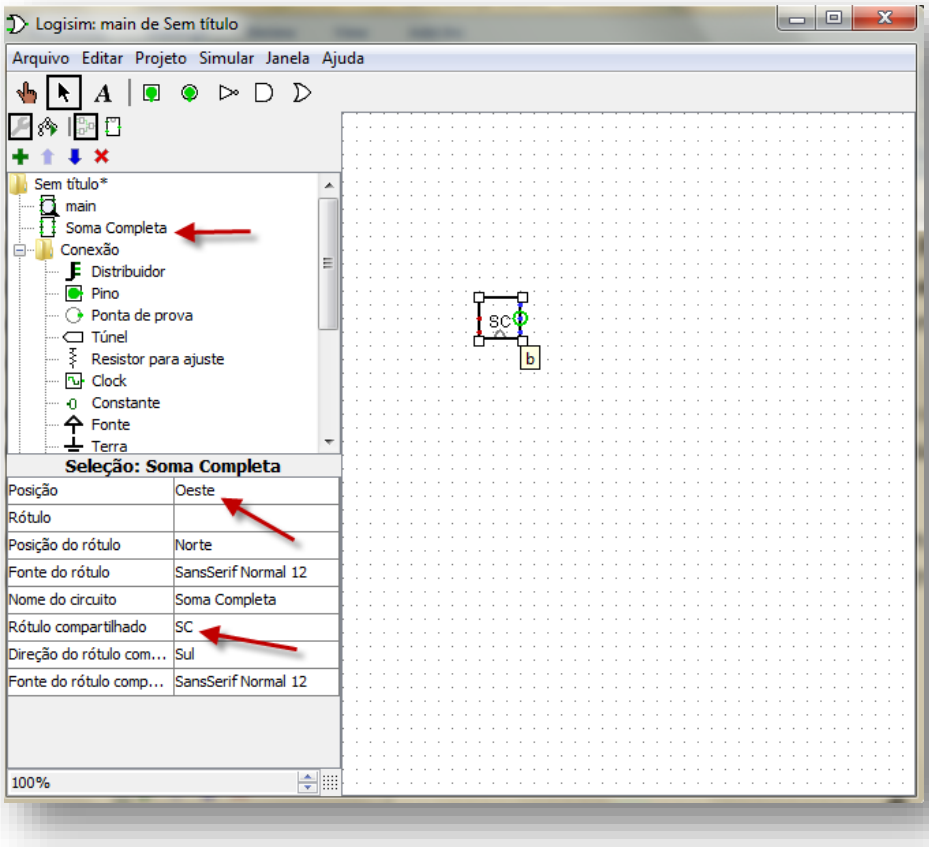


Figura 100: Colocando um sub-circuito de Soma Completa como um bloco no circuito principal. Esta tela foi capturada com o mouse sobre a entrada b do sub-circuito

Para obter exatamente o que está mostrado na Figura 100, ainda será necessário usar o painel de atributos para:

- mudar a posição do sub-circuito para Oeste, fazendo com que suas entradas fiquem à direita do bloco, e as saídas à esquerda, o que é mais conveniente para o desenho do somador de 4 bits;
- colocar “SC” como rótulo compartilhado dos sub-circuitos de Soma Completa.

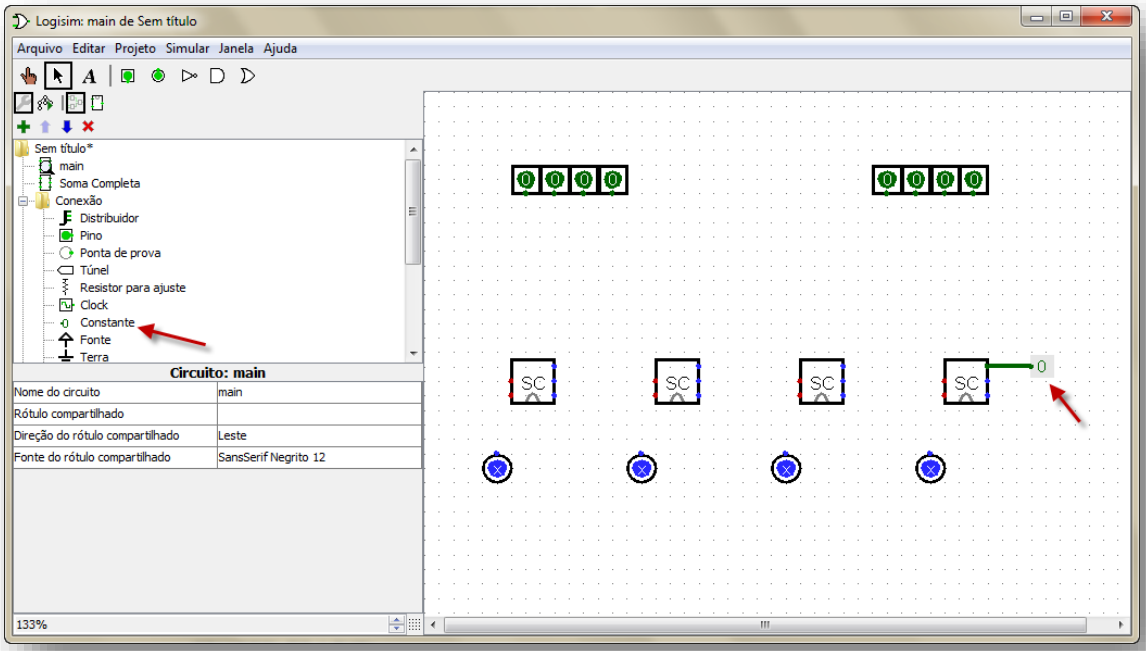


Figura 101: Iniciando o circuito somador de 4 bits, com os componentes de soma completa, entradas e saídas, e a constante 0 na entrada v do bit menos significativo

Como você pode ver, subimos de nível. No projeto do circuito somador de 4 bits, detalhes do sub-circuito Soma Completa, importantes para o seu projeto interno, podem ser esquecidos – o que interessa agora é somente sua funcionalidade.

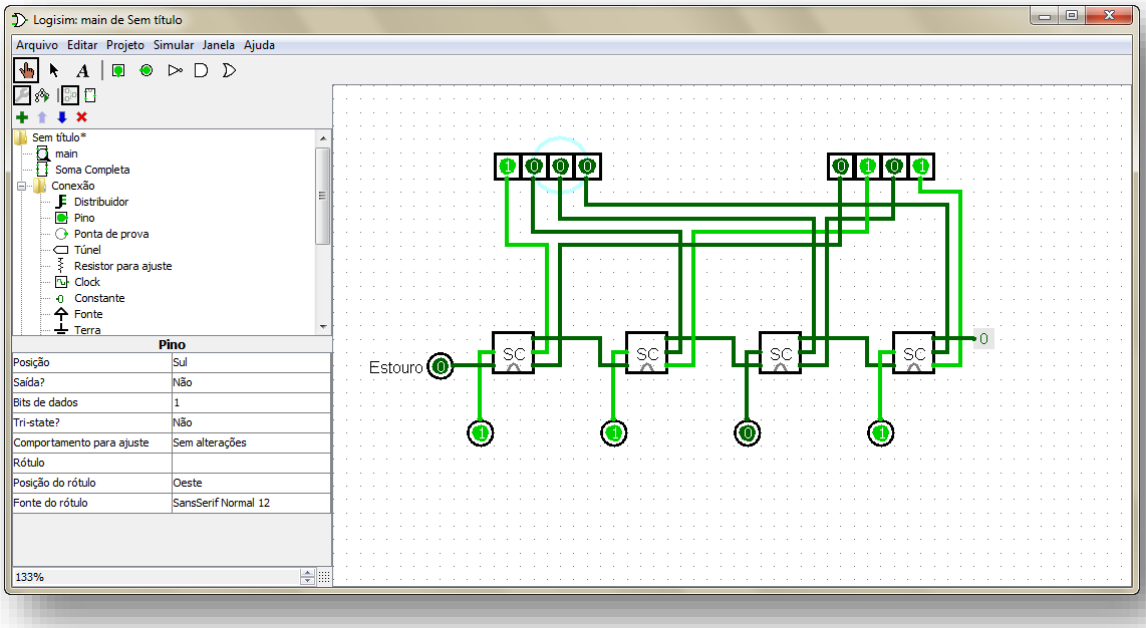


Figura 102: Somador de 4 bits completo, utilizando sub-circuitos de soma completa

A Figura 102 mostra o circuito completo do somador de 4 bits. Compreender este circuito é certamente muito mais fácil do que entender o somador da Figura 93.

2.8 Síntese de um display hexadecimal

Como um exemplo mais detalhado de síntese automática de circuitos combinatórios a partir de sua tabela da verdade, vamos o usar o Logisim para construir um display hexadecimal, isto é, um circuito que "acende" os filamentos correspondentes ao símbolo hexadecimal codificado em seus 4 bits de entrada em uma lâmpada de 7 segmentos. Este é um dispositivo simples e eficaz para a visualização de algarismos decimais e de algumas letras, que você certamente já viu em relógios, elevadores ou sintonizadores de TV a cabo.

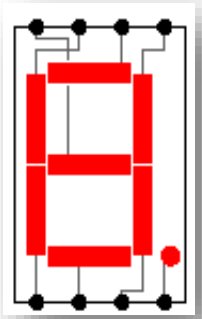


Figura 103: Uma lâmpada de 7 segmentos com todos os filamentos acesos

O Logisim oferece este componente, na biblioteca *Entrada/Saída*. Cada um dos pinos acente um dos filamentos, com exceção de um pino que acende um ponto decimal, que não vamos usar aqui.

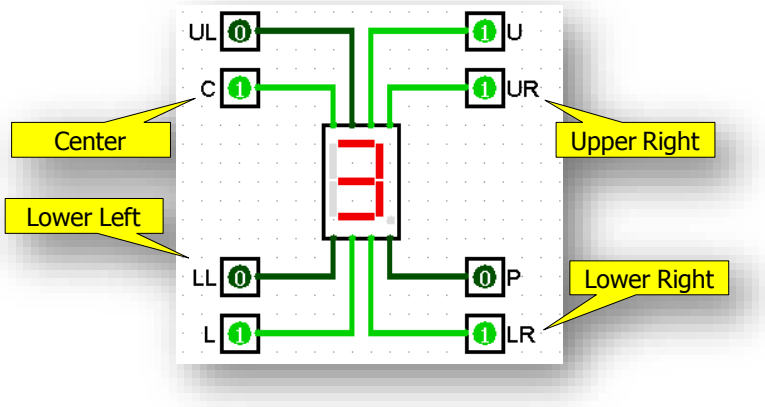


Figura 104: Nomes dos pinos em uma lâmpada de 7 segmentos

A Figura 104 mostra os rótulos dos pinos de entrada de uma lâmpada de 7 segmentos, e na Figura 105 nós vemos o arranjo de entradas e saídas do circuito que iremos sintetizar (o display hexadecimal já existe também pronto no Logisim, mas vamos reconstruí-lo).

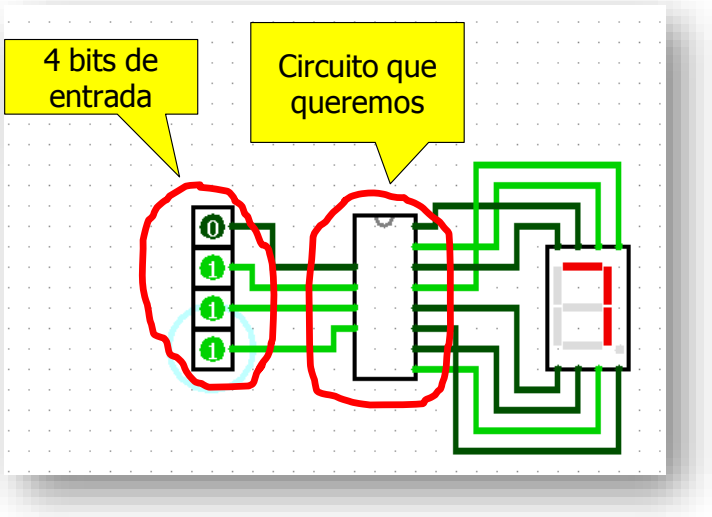


Figura 105: Arranjo de entradas e saídas de circuito de controle de um display hexadecimal

Para isso, abrimos a janela *Análise Combinacional* do Logisim, e construímos uma tabela da verdade com 4 entradas, **a3**, **a2**, **a1** e **a0**, e com 7 saídas, **UL**, **U**, **C**, **UR**, **LL**, **P**, **L** e **LR**. Para cada linha colocamos 1 nas saídas correspondentes aos filamentos que, acesos, compõem o dígito hexadecimal formado pelos bits de entrada. A tabela final está mostrada na Figura 106.

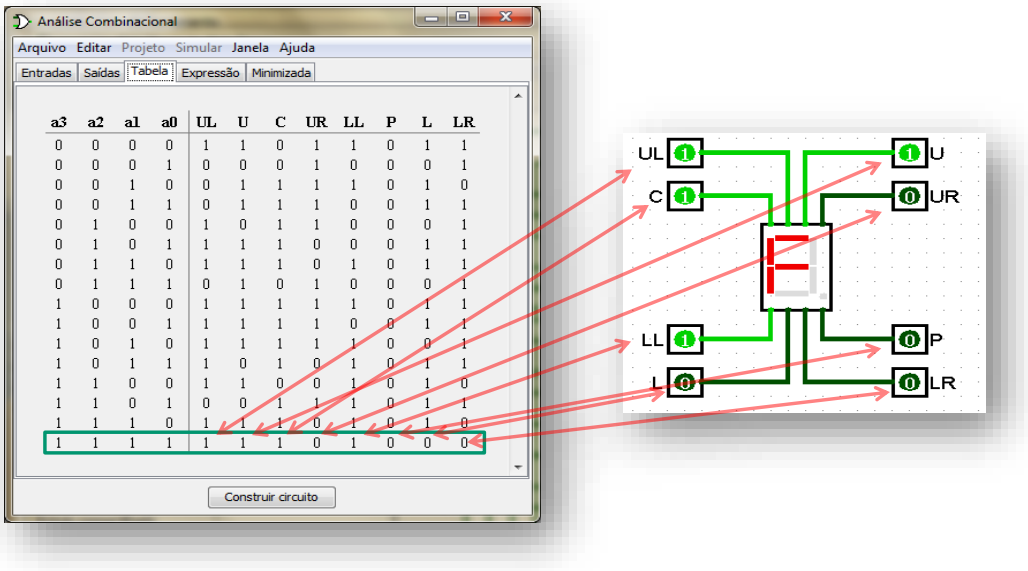


Figura 106: A tabela da verdade completa para o display hexadecimal, com destaque para a determinação das saídas para formar um F

Feito isso, basta apertar *Construir circuito* para obter o circuito da Figura 107. O circuito parece complicado? Pode ser, mas isso não é um problema. O circuito foi construído automaticamente, a partir de uma tabela da verdade, usando algoritmos do Logisim que são melhorias do método que vimos para obtenção de somas canônicas. Isso nos garante a sua correção. E ele pode ser usado como um módulo, esquecendo completamente seus detalhes, como fizemos na Figura 105.

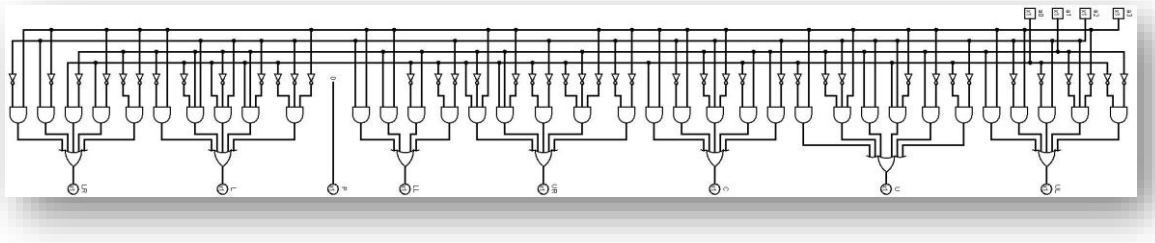


Figura 107: Circuito gerado automaticamente pelo Logisim para o controlador de display hexadecimal

2.9 Comparação de binários sem signal

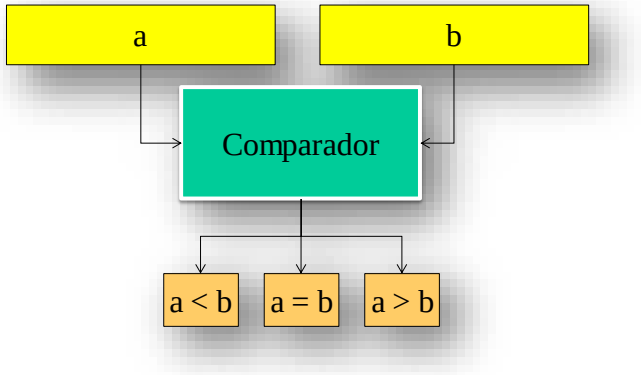


Figura 108: Entradas e saídas de um circuito comparador

Vimos na seção 2.5 que circuitos lógicos podem realizar operações aritméticas. Vamos agora atacar o problema de construir circuitos lógicos que permitam comparar duas variáveis a e b de, digamos, 32 bits cada uma, codificadas como binários sem sinal. A saída do circuito deve indicar se $a = b$, $a < b$ ou $a > b$, como mostrado na Figura 108.

Aqui também temos problemas com o uso do método de cobertura de 1s na tabela da verdade, que teria as mesmas 2^{64} linhas da tabela do somador de 32 bits. Vamos aplicar o mesmo enfoque que usamos na soma: verificar como fazemos a comparação, e procurar resolver o problema por etapas que seriam feitas por módulos menores.

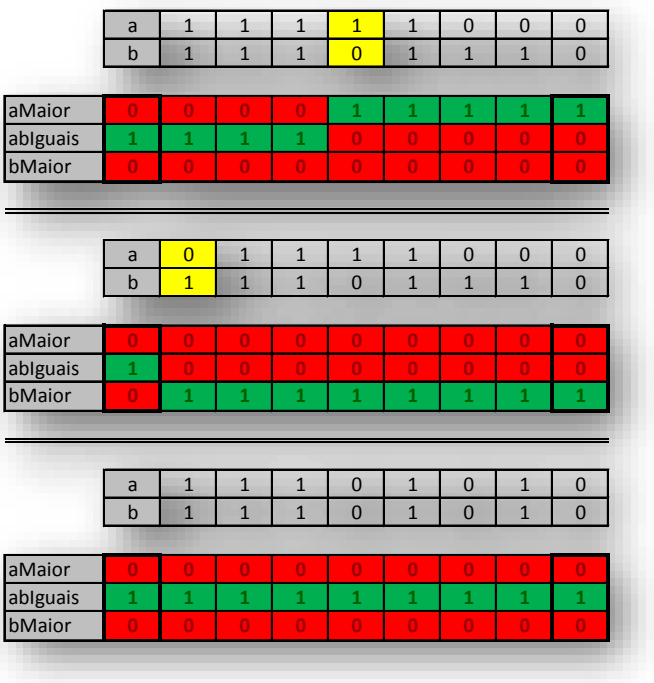


Figura 109: Casos de comparação entre as entradas a e b.

É fácil ver que, ao comparar dois binários de mesmo tamanho, sem sinal, devemos comparar bit a bit, começando com o bit mais significativo. Na primeira diferença, já podemos concluir que a parcela com o bit igual a 1 é definitivamente a maior, e os bits restantes, que são menos significativos, não interessam para o resultado final. A Figura 109 mostra três casos de comparação de entradas a e b, cada uma com 8 bits. No primeiro caso as entradas diferem no quarto bit mais significativo, e $a > b$. No segundo caso as entradas já diferem no primeiro bit mais significativo, e $b > a$. No terceiro caso as entradas são iguais.

Para construir um circuito que faça a comparação de binários sem sinal nós precisamos de um comparador de 1 bit que leve em consideração a possibilidade do resultado já ter sido estabelecido por algum bit mais significativo.

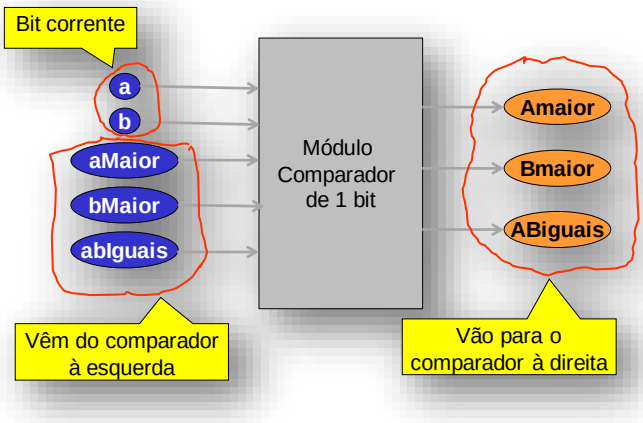


Figura 110: Entradas e saídas para um módulo comparador de 1 bit

Vemos na Figura 110 um esquema de entradas e saídas para o módulo comparador de 1 bit. As entradas **a** e **b** vêm do bit em questão das variáveis que estamos comparando. Da comparação

já feita com bits mais significativos que o bit em questão vêm as entradas **aMaior**, **bMaior** e **abIguais**. A Figura 111 mostra o arranjo de módulos que compõem um comparador de 4 bits.

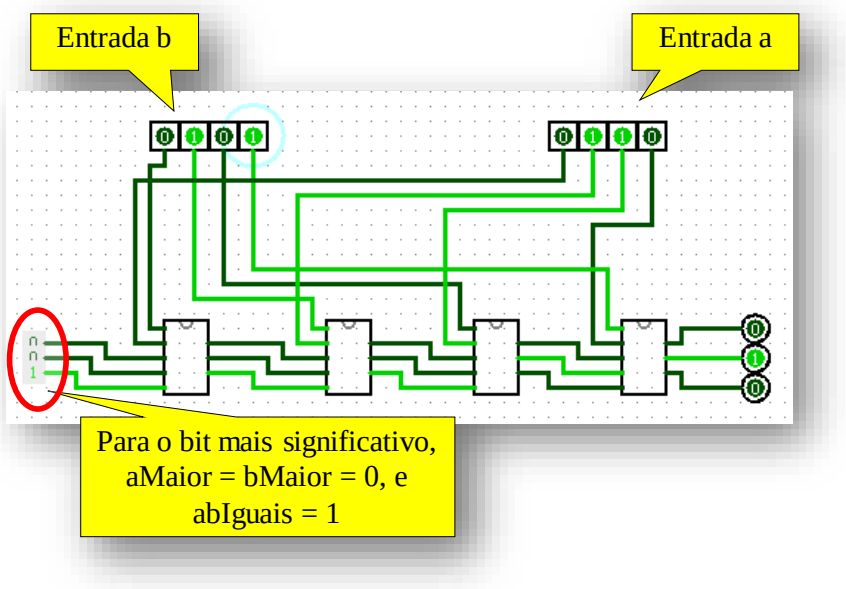
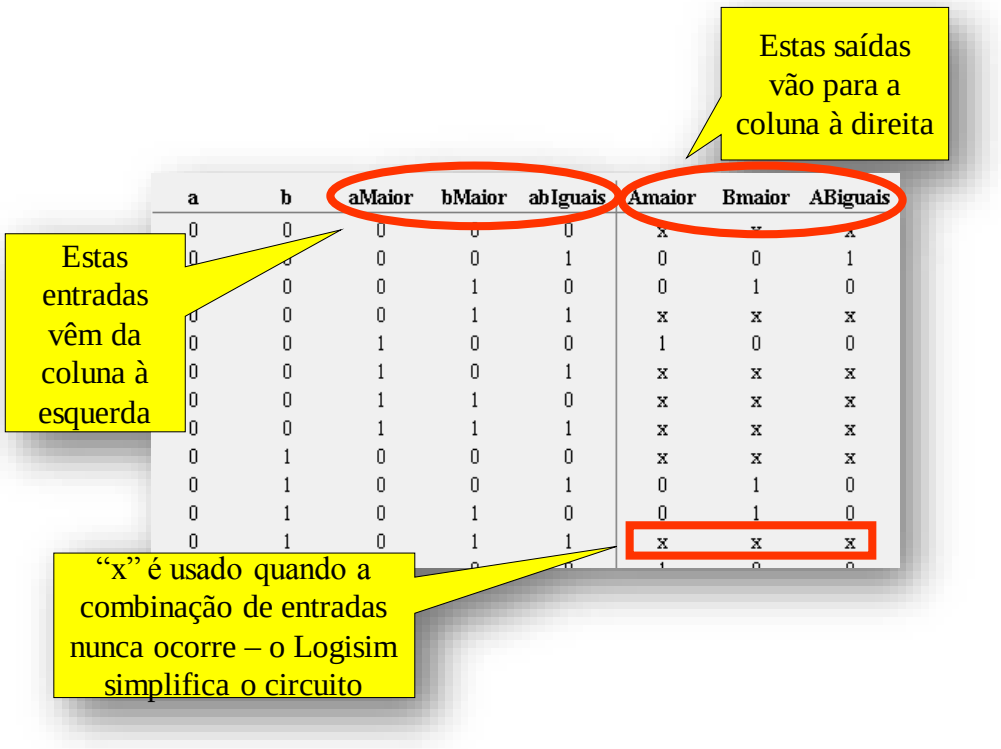


Figura 111: Arranjo de módulos para um comparador de 4 bits

Temos um módulo comparador para cada bit das variáveis de entrada. Cada módulo comparador tem cinco entradas:

- Duas são os bits das entradas da coluna correspondente ao módulo;
- As três outras são designadas por **aMaior**, **bMaior** e **abIguais**, e, exceto para o módulo mais à esquerda, são produzidas pelo módulo comparador à esquerda do módulo em questão. Elas têm o resultado da comparação já realizada com os bits mais significativos e portanto, uma e somente uma dessas entradas poderá ter o valor 1. Para o módulo mais à esquerda, que corresponde ao bit mais significativo, nós impomos o valor 1 para a entrada **abIguais**, e 0 para **aMaior** e **bMaior.**

A Figura 112 mostra as primeiras linhas de uma tabela da verdade para o módulo comparador de 1 bit; a tabela completa tem $2^5 = 32$ linhas. A coluna **aMaior** é uma entrada desse módulo, enquanto a coluna **Amaior** é uma saída. A mesma convenção é usada para **bMaior**, **Bmaior**, **abIguais** e **ABiguais**. Combinações onde mais de uma dentre as variáveis **aMaior**, **bMaior** e **abIguais** têm o valor 1 ou em que todas as três são iguais a 0 nunca devem ocorrer, e o símbolo x é usado na tabela para indicar que os valores das saídas não têm interesse nesses casos. O Logisim tira proveito disso para obter circuitos mais simples.


Estas saídas vão para a coluna à direita
Estas entradas vêm da coluna à esquerda
"x" é usado quando a combinação de entradas nunca ocorre – o Logisim simplifica o circuito

a	b	aMaior	bMaior	abIguais	Amaior	Bmaior	ABiguais
0	0	0	0	0	x	x	x
0	0	0	0	1	0	0	1
0	0	0	1	0	0	1	0
0	0	0	1	1	x	x	x
0	0	1	0	0	1	0	0
0	0	1	0	1	x	x	x
0	0	1	1	0	x	x	x
0	0	1	1	1	x	x	x
0	1	0	0	0	x	x	x
0	1	0	0	1	0	1	0
0	1	0	1	0	0	1	0
0	1	0	1	1	x	x	x

Figura 112: Tabela da verdade para um comparador de 1 bit

A Figura 113 mostra o circuito comparador de 1 bit obtido dessa tabela da verdade.

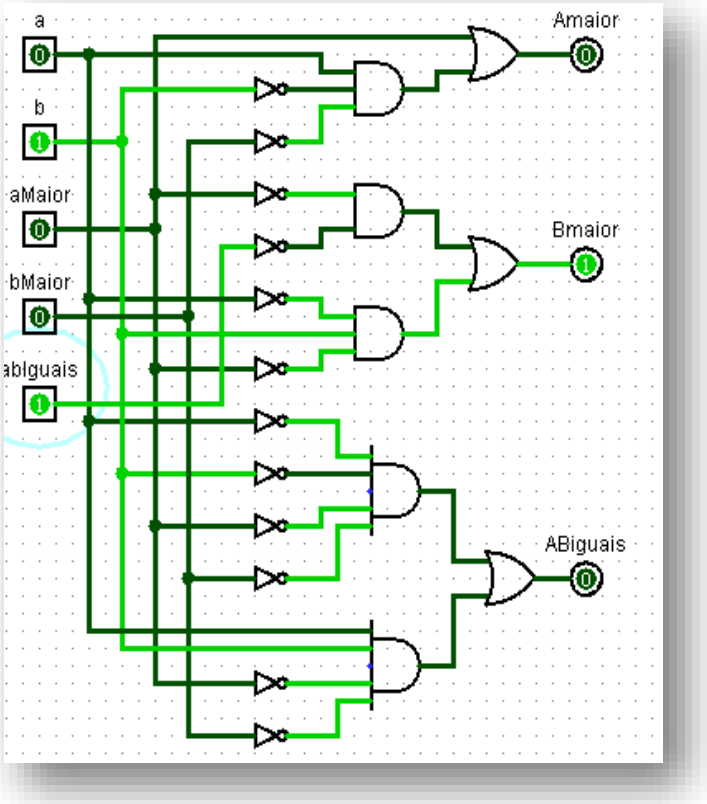


Figura 113: Circuito comparador de 1 bit

2.10 Multiplexadores, Demultiplexadores e Decodificadores

Conduzir bits de um lugar para outro é essencial para o funcionamento de um computador, para conduzir dados a circuitos capazes de, por exemplo, efetuar operações aritméticas. Nós veremos agora *multiplexadores* e *demultiplexadores*, que são circuitos que conectam entradas e saídas de acordo com um *endereço* – e o endereço também é uma entrada.

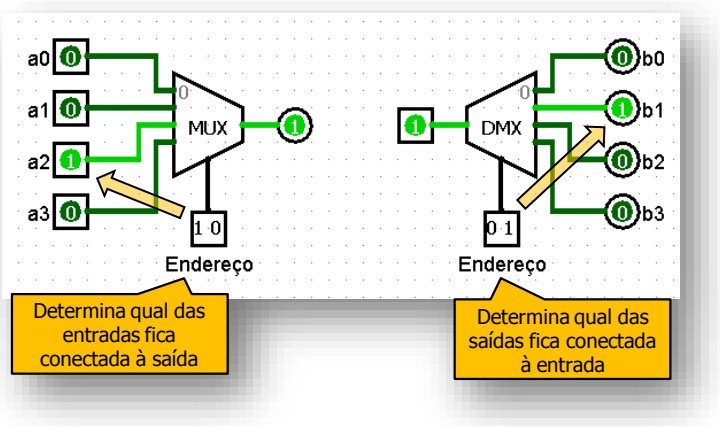


Figura 114: Multiplexadores (MUX) e Demultiplexadores (DMX)

Como mostra a Figura 114,

- Um *multiplexador* conecta uma única entre várias entradas de dados a um destino; a entrada escolhida é designada por um *endereço*;
- Um *demultiplexador* conecta uma entrada de dados a um dentre vários destinos; o destino escolhido é designado por um *endereço*.

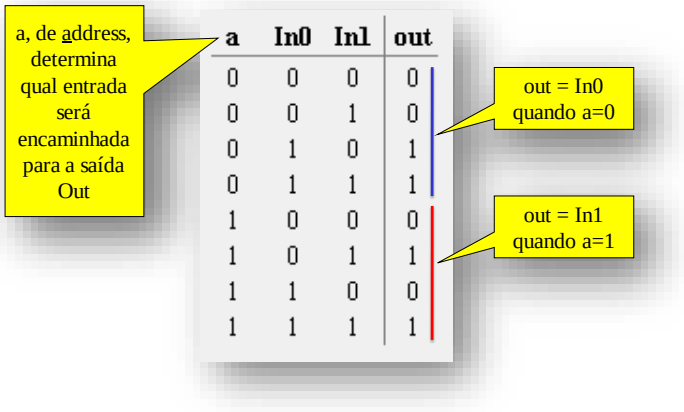
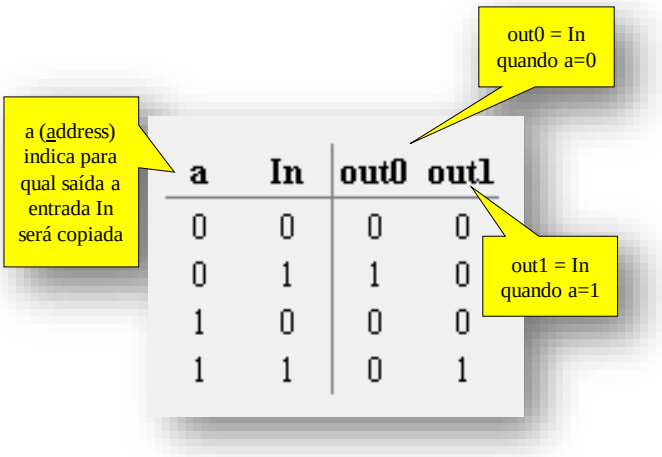

a, de address, determina qual entrada será encaminhada para a saída Out
a In0 In1 out
0 0 0 0
0 0 1 0
0 1 0 1
0 1 1 1
1 0 0 0
1 0 1 1
1 1 0 0
1 1 1 1
out = In0 quando a=0
out = In1 quando a=1

Figura 115: Tabela da verdade para um multiplexador de 1 bit

A Figura 115 mostra uma tabela da verdade para um multiplexador de 1 bit de endereço. Esse circuito tem três entradas: **In0** e **In1**, que são as variáveis fonte de informação, e o endereço **a**, que decide qual dentre **In0** e **In1** será conectada à saída **out**.



a	In	out0	out1
0	0	0	0
0	1	1	0
1	0	0	0
1	1	0	1

Figura 116: Tabela da verdade para um demultiplexador de 1 bit

Na Figura 116 nós vemos a tabela da verdade para um demultiplexador de 1 bit. Esse circuito tem as entradas **a** e **In**. A entrada **a** é um endereço de 1 bit e que designa qual das saídas, **out0** ou **out1**, será conectada à entrada fonte de informação **In**.

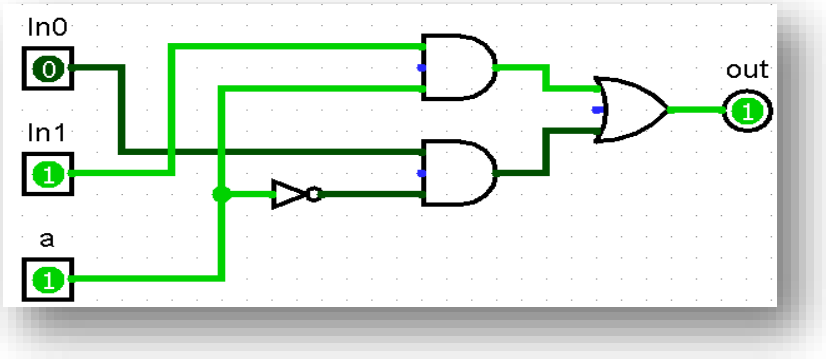


Figura 117: Circuito multiplexador de 1 bit

Multiplexadores de 1 bit também podem ser usados como módulos para a construção de multiplexadores de mais bits, com o arranjo mostrado na Figura 118.

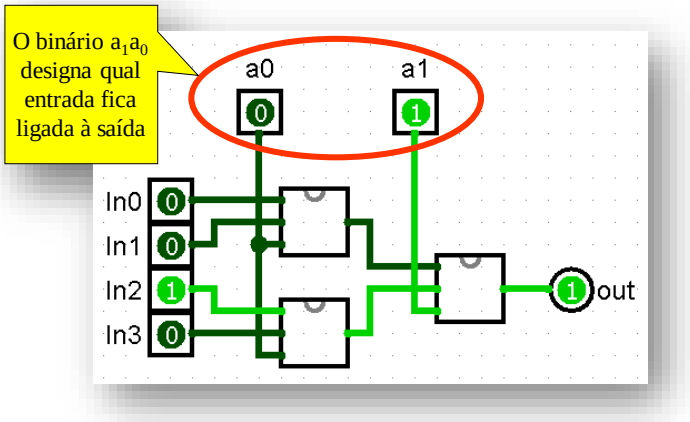


Figura 118: Um multiplexador de 2 bits obtido usando multiplexadores de 1 bit

A Figura 119 mostra o circuito demultiplexador de 1 bit obtido a partir da tabela da verdade da Figura 116.

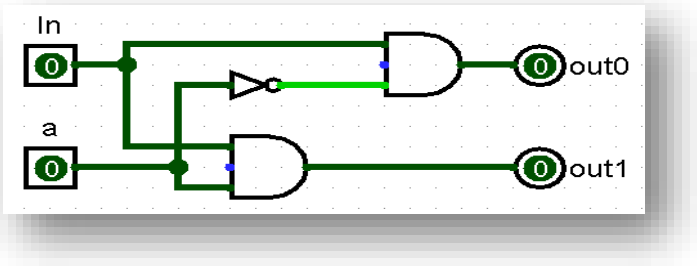


Figura 119: Circuito demultiplexador de 1 bit

Demultiplexadores com uma "largura de endereço" (número de bits de endereço) maior também podem ser obtidos com arranjos hierárquicos similares, como mostra a Figura 120.

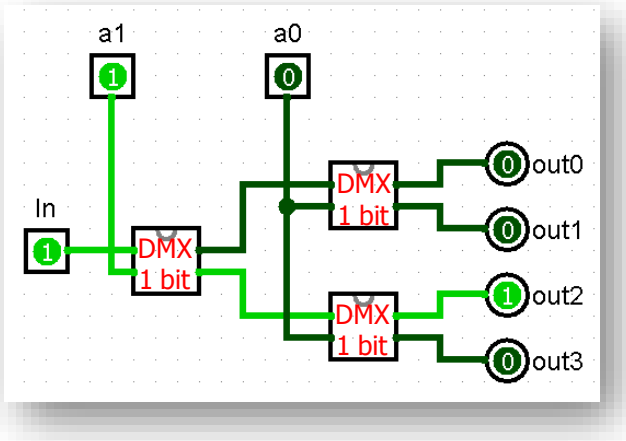


Figura 120: Circuito demultiplexador de 2 bits, obtido a partir de demultiplexadores de 1 bit

Um *decodificador* é um circuito com n entradas de 1 bit, e 2^n saídas de 1 bit. Destas, uma e apenas uma terá o valor 1 – aquela cujo endereço é a decodificação do valor codificado como binário sem sinal nas entradas. Um decodificador pode ser obtido usando um demultiplexador, como mostra a Figura 121.

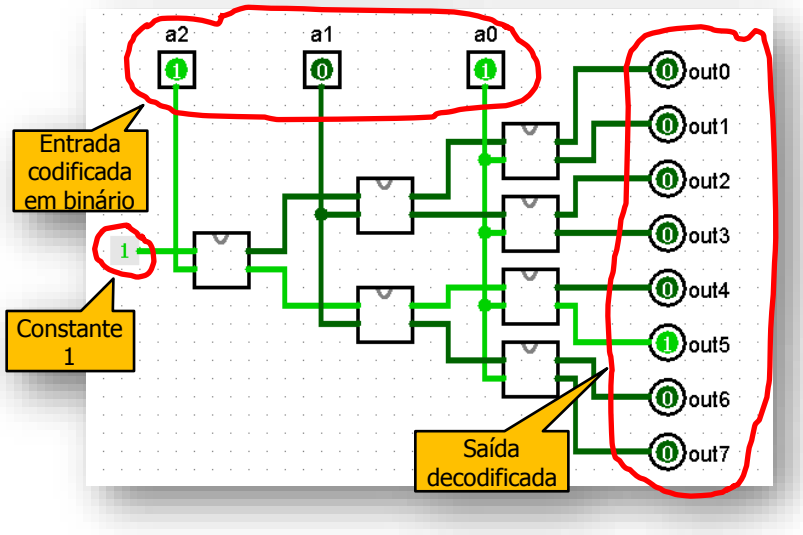


Figura 121: Decodificador de 3 bits

3 Circuitos Sequenciais

Nós vimos que circuitos combinatórios podem, com arranjos adequados, fazer operações aritméticas ou comparações com operandos de 32 ou 64 bits. Mas seria possível construir um circuito combinatório para realizar a soma de, digamos, um milhão de parcelas de 32 bits cada? Teoricamente sim, mas na prática isso não é feito – não somente por dificuldades técnicas, mas também porque é muito mais simples realizar esta soma parcela por parcela, acumulando resultados parciais em alguma memória. Para isso são utilizados *circuitos seqüenciais*, que diferem dos circuitos combinatórios por serem capazes de armazenar dados. Em um dado instante as saídas de um circuito sequencial não dependem apenas dos valores correntes de suas entradas, como nos circuitos combinatórios, mas são também funções de valores armazenados em suas memórias.

3.1 Flip-flops e Registradores

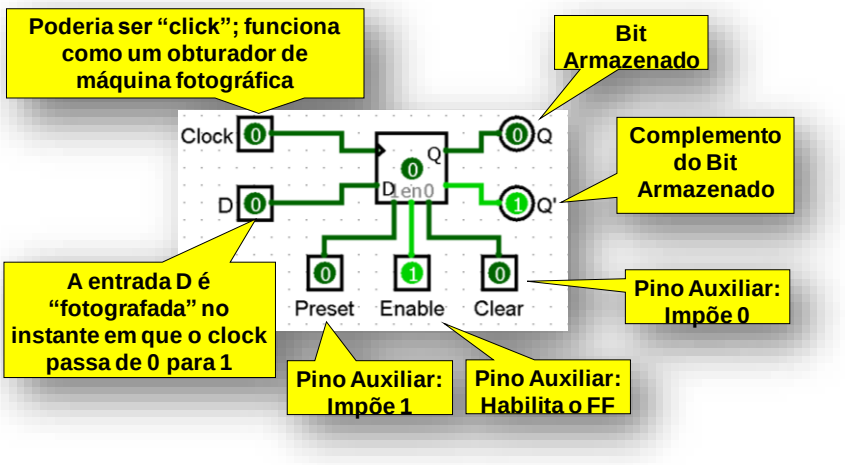


Figura 122: Flip-flop tipo D

O circuito seqüencial mais básico é conhecido pelo nome em inglês de *flip-flop*, e, como sempre começamos pelos casos mais simples, um flip-flop é capaz de armazenar um único bit. Existem vários tipos de flip-flop, mas aqui nós só veremos flip-flops do *tipo D*. A Figura 122 mostra um flip-flop tipo D disponível no Logisim, e que possui 5 pinos:

- Pino **D**: é onde fica a informação – o *D*ado de entrada – que pode vir a ser armazenada no flip-flop;
- Pino **Clock**: é um pino que funciona como um obturador de uma máquina fotográfica. No exato instante em que o sinal aplicado ao *Clock* passa de 0 para 1, o flip-flop passa a armazenar o valor corrente do pino *Input*. O nome "clock" vem do fato deste sinal frequentemente se originar de uma fonte de tempo, como veremos mais tarde. "Click" seria um nome mais adequado para a analogia com a máquina fotográfica.
- Pino **Q**: é uma saída que tem o valor armazenado no flip-flop;
- Pino **Q'**: é uma saída que tem o complemento do valor armazenado no flip-flop;
- Pinos **Set** e **Reset**: são entradas auxiliares que facilitam a imposição de um valor para o flip-flop. Essas entradas são normalmente usadas para inicialização ou re-inicialização do flip-flop.

- Pino **Enable**: é uma entrada auxiliar que serve para habilitar ou inibir o comportamento do flip-flop.

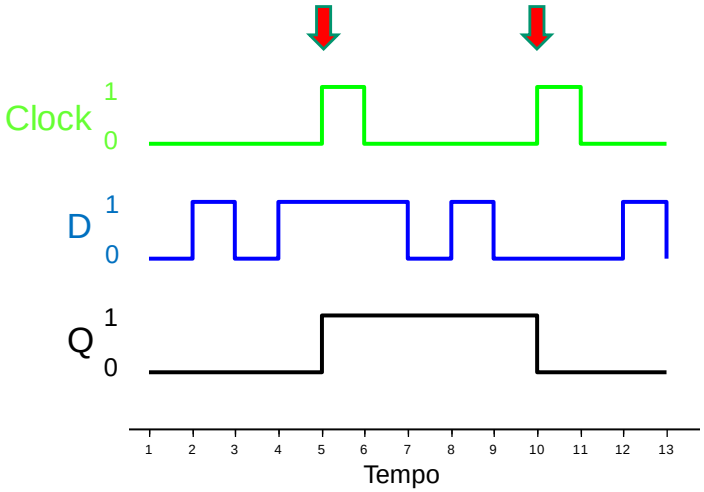


Figura 123: Carta de tempo para um flip-flop tipo D. As setas destacam os instantes de subida do clock, quando a entrada D é copiada pelo flip-flop.

A Figura 123 mostra um gráfico – uma *carta de tempo* – que ilustra um exemplo de evolução temporal de um flip-flop tipo D, onde:

- no instante 1 as entradas **D** e **Clock** valem 0, assim como a saída **Q**;
- nos instantes 2, 3 e 4 respectivamente a entrada **D** muda de 0 para 1, de 1 para 0 e de 0 para 1 novamente, sem que isso afete a saída **Q**, pois a entrada **Clock** permanece em 0 durante esse intervalo;
- no instante 5 a entrada **Clock** sobe, mudando de 0 para 1. É a este sinal que o flip-flop reage, copiando ("fotografando") a entrada **D**. Com isso o bit armazenado muda também de 0 para 1;
- no instante 6 a entrada **Clock** desce, mas isso não afeta o estado do flip-flop;
- nos instantes 7, 8 e 9 a entrada **D** oscila novamente, sem afetar o estado do flip-flop;
- no instante 10 o sinal do **Clock** sobe, e a saída **Q** passa para 0, copiando o valor de **D** nesse instante;

E por aí vai.

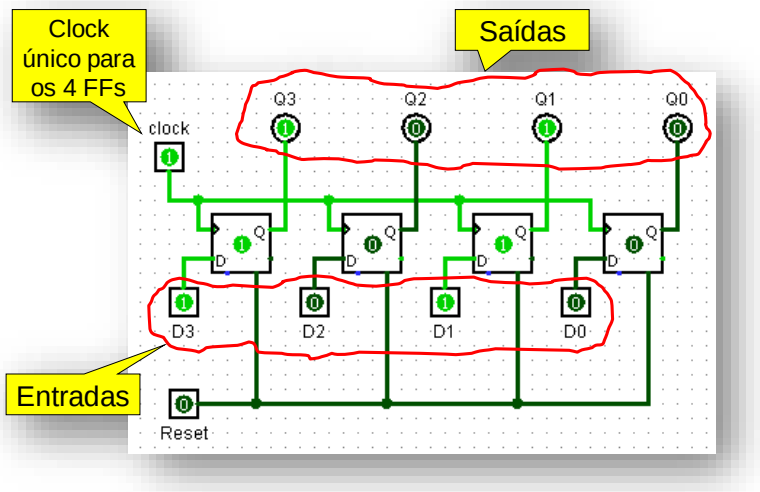


Figura 124: Um registrador de 4 bits formado por flip-flops tipo D

Flip-flops podem ser agrupados formando *registradores*, que são conjuntos de flip-flops com alguma missão em comum. A Figura 124 mostra um registrador de 4 bits, cuja função é similar à de um único flip-flop, mas com capacidade para armazenamento de 4 bits. Ele é composto por flip-flops do tipo D, e podemos reparar na figura que:

- um mesmo sinal de clock comanda os 4 flip-flops;
- na subida deste sinal, isto é, quando o clock passa de 0 para 1, as quatro entradas D são copiadas para os flip-flops;
- uma entrada **Reset** coloca 0 em todos os flip-flops, ao receber um sinal 1.

O Logisim oferece uma série de circuitos já prontos, que encontram-se armazenados em bibliotecas, e que podem ser utilizados como peças para a montagem de circuitos maiores, da mesma forma como já usamos portas lógicas. Flip-flops tipo D, outros tipos de flip-flop, registradores e vários outros componentes se encontram na biblioteca *Memória*.

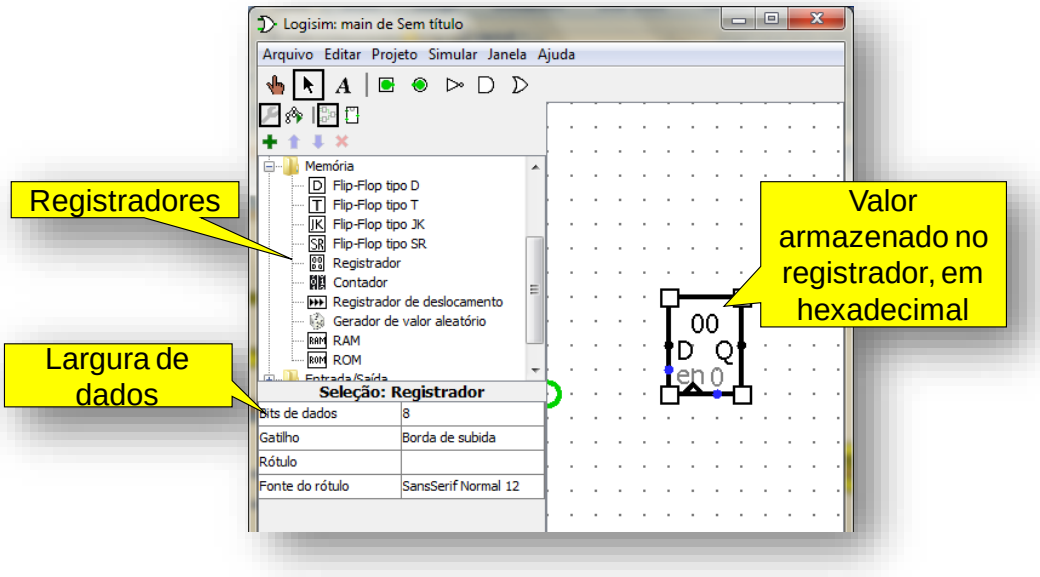


Figura 125: Um registrador da biblioteca *Memória* do Logisim, com largura de 8 bits

Registradores do Logisim são sempre apresentados como um único bloco que se assemelha a um flip-flop, mas que é capaz de armazenar um número de bits à escolha do usuário. Um cabo conectado à entrada D de um registrador de 4 bits deve também ter uma “largura” de 4 bits.

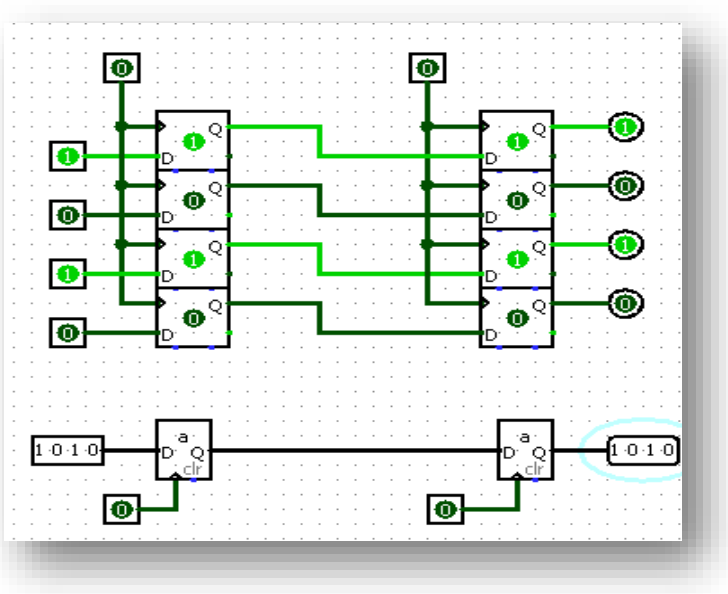


Figura 126: Dois diagramas de um mesmo circuito. No circuito de cima, fios e registradores têm 1 bit de largura; no de baixo, 4 bits

Na Figura 126 nós vemos dois diagramas para um mesmo circuito com dois registradores de 4 bits conectados. O diagrama de cima utiliza somente elementos de largura de 1 bit, com os quais já estamos familiarizados. O de baixo utiliza entradas, saídas, registradores e cabos de 4 bits de largura. Seu desenho é por isso mesmo muito mais simples e, portanto, de mais fácil compreensão. A largura de bits de componentes como registradores, entradas e saídas é controlada pelo usuário, usando o campo "*Bits de Dados*" no painel de atributos, como na Figura 125. O Logisim facilita a nossa vida dando a cabos a largura de bits dos componentes aos quais o cabo se conecta, e alertando o usuário nos casos de incompatibilidade.

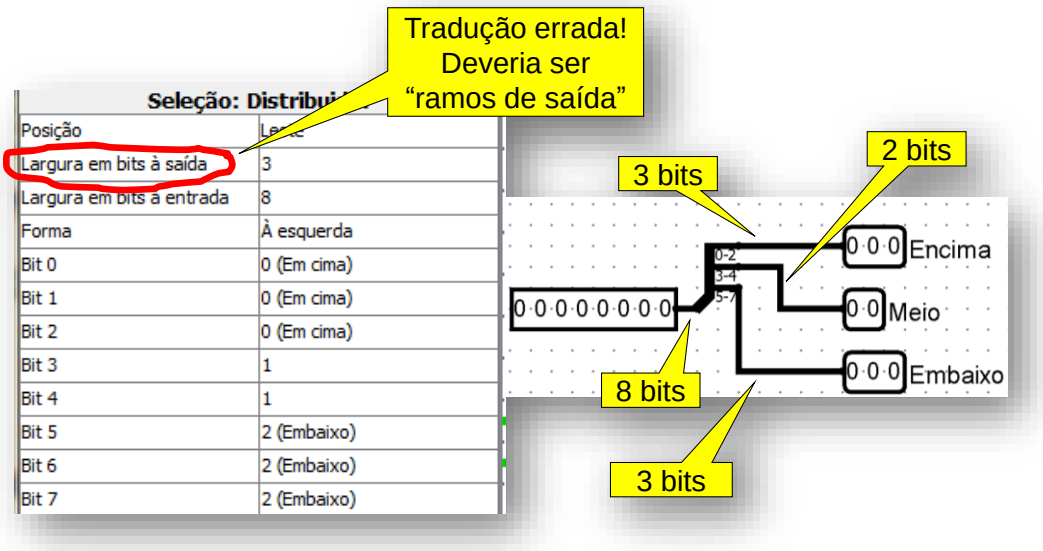


Figura 127: Uma bifurcação de um cabo de 8 bits em um de 2 bits e outros dois de 2 bits de largura. O retângulo à esquerda é uma ampliação do painel de atributos da bifurcação

O Logisim oferece ainda bifurcações (*splitters*) que permitem dirigir os bits de um cabo com largura maior para outros de largura menor, como mostrado na Figura 127.

3.2 Barramentos e Controle de Fluxo de Dados

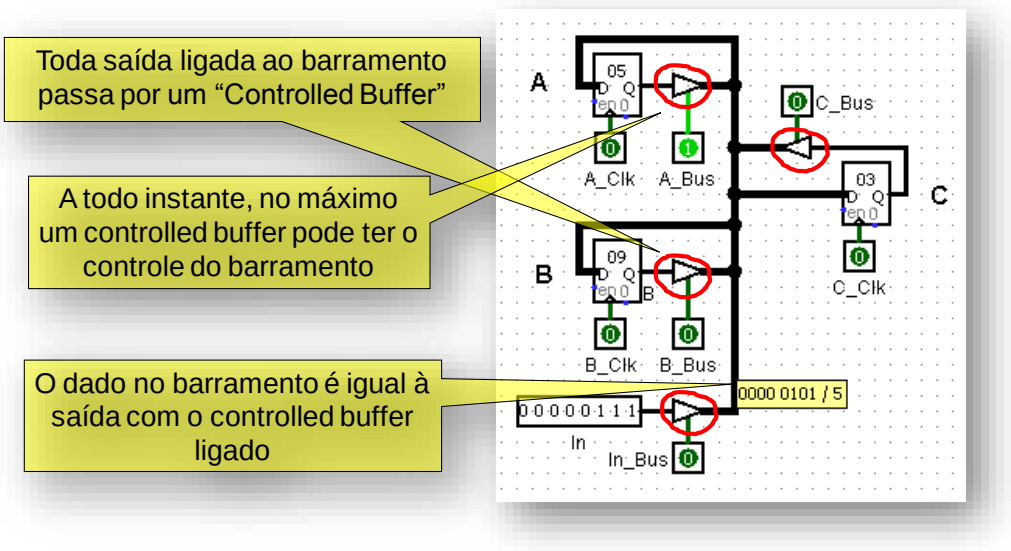


Figura 128: Um barramento conectando uma entrada de 8 bits e três registradores

Nos circuitos que vimos até agora um cabo só pode ser usado para conectar uma saída de um componente a uma ou mais entradas de outros componentes. Essa restrição vem por um lado da falta de sentido lógico nesse tipo de conexão: se uma das saídas tem 0 e outra 1, qual valor fica no barramento? Por outro lado, vem também dos circuitos reais: se uma de duas saídas conectadas a um cabo tem o valor 0 e outra tem o valor 1, temos uma voltagem alta ligada diretamente a uma voltagem baixa, ou seja, um curto-circuito.

Um componente especial, o *buffer controlado*, permite fazer esse tipo de ligação, o que simplifica muito o desenho de circuitos. Diversas saídas podem ser conectadas a um único cabo se essas conexões passarem por um *buffer controlado*. Este cabo compartilhado recebe o nome de *barramento*. Todo *buffer controlado* tem, como o próprio nome indica, um pino de controle que abre ou fecha a conexão com o barramento. O projetista de um circuito deve cuidar para que, a qualquer instante, no máximo um dentre todos os *buffer controlados* ligados a um mesmo barramento esteja aberto.

Colocar 7 no registrador **A**		
Passo	Sinal	Comentário
1	**In** = 7	Colocar 7 (em binário, 111) na entrada **In**
2	**In_Bus** = 1	Colocar o valor de **In** no barramento
3	**A_Clk** = 1	Copiar o valor do barramento no registrador **A**
4	**A_Clk** = 0	Zerar o clock do registrador **A**
5	**In_Bus** = 0	Liberar o barramento

Colocar 3 no registrador B		
Passo	Sinal	Comentário
1	**In** =3	Colocar 3 (em binário, 11) na entrada **In**
2	**In_Bus** = 1	Colocar o valor de **In** no barramento
3	**B_Clk** = 1	Colocar o valor do barramento no registrador **B**
4	**B_Clk** = 0	Zerar o clock do registrador **B**
5	**In_Bus** = 0	Liberar o barramento

Colocar no registrador **C** o conteúdo do registrador **A**		
Passo	Sinal	Comentário
1	**A_Bus** = 1	Colocar o valor do registrador **A** no barramento
2	**C_Clk** = 1	Copiar o valor do barramento no registrador **C**
3	**C_Clk** = 0	Zerar o clock do registrador **C**
4	**A_Bus** = 0	Liberar o barramento

Figura 129 : Exemplos de fluxos de dados realizáveis pelo circuito da Figura 128.

O circuito da Figura 128 permite que um dado na entrada **In** seja copiado por qualquer dos registradores **A**, **B** ou **C**, e permite também que o valor em qualquer registrador seja copiado por qualquer um dos outros registradores. Estes fluxos são controlados pelos sinais de clock dos registradores e de controle dos buffers controlados. Dados são transferidos de um ponto para outro ligando e desligando esses sinais em uma sequência apropriada para a transferência desejada.

A Figura 129 mostra exemplos de sequências de sinais que devem ser aplicados ao circuito para que ocorram transferências de dados específicas. Os sinais devem ser aplicados na sequência dos passos. Na figura usamos a notação (por exemplo) **B_Clk** = 1 para indicar que o valor 1 deve ser aplicado à entrada **B_Clk** pelo operador Logisim, e **In** = 3 para dizer que o valor 3 (em binário, 11) deve ser aplicado à entrada **In**.

3.3 Memórias

O Logisim oferece memórias RAM (*Random Access Memory*) e ROM (*Read Only Memory*) como componentes de sua biblioteca *Memory*. Memórias armazenam informações como conjuntos de bits chamados *palavras*; cada palavra possui um endereço na memória. Uma memória tem como atributos sua largura de dados, isto é, o número de bits em cada palavra da memória, e a largura do endereço. Com n bits de endereço uma memória tem no máximo 2^n palavras. No Logisim, a largura de bits do endereço determina também o tamanho da memória: uma memória de n bits de endereço tem exatamente 2^n palavras.

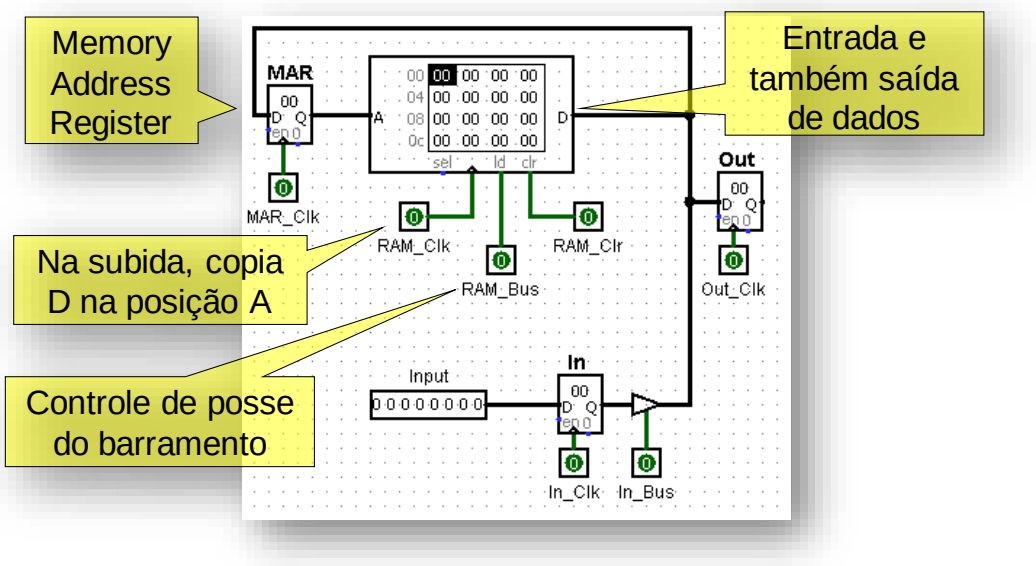


Figura 130: Uma memória RAM do Logisim em um arranjo com registradores e barramento

A Figura 130 mostra um arranjo de uma memória RAM e de registradores em torno de um barramento. As operações de leitura e escrita se fazem com uma única palavra da memória, determinada pelo valor aplicado à entrada **A**:

- para escritas, o sinal **RAM_Clk** funciona como o clock de um registrador: na subida de 0 para 1, o valor presente na entrada **D** é copiado para a posição de memória endereçada pela entrada A, o que, no arranjo da Figura 130, é o valor armazenado no registrador **MAR**, o *Memory Addres Register*;
- para leituras, o sinal **RAM_Bus** funciona como um buffer controlado conectado à saída de um registrador: enquando seu valor for 1, a memória coloca no barramento o conteúdo da palavra endereçada pela entrada **A**.

Escrever 9 na posição de memória 15		
Passo	Sinal	Comentário
1	**Input** = 15	Colocar 15 (em binário, 1111) na entrada **Input**
2	**In_Clk** = 1	Copiar no registrador **In** a entrada **Input**
3	**In_Clk** = 0	Zerar o clock do registrador **In**
4	**In_Bus** = 1	Colocar o valor de **In** no barramento
5	**MAR_Clk** = 1	Copiar o valor do barramento no registrador **MAR** (e também na entrada **A** da RAM)
6	**MAR_Clk** = 0	Zerar o clock do registrador **MAR**
7	**In_Bus** = 0	Liberar o barramento
8	**Input** = 9	Colocar 9 (em binário, 1001) na entrada **Input**
9	**In_Clk** = 1	Copiar no registrador In a entrada **Input**
10	**In_Clk** = 0	Zerar o clock do registrador **In**
11	**In_Bus** = 1	Colocar o valor de **In** no barramento
12	**RAM_Clk** = 1	Copiar o valor do barramento no endereço **A** da RAM
13	**RAM_Clk** = 0	Zerar o clock da memória
14	**In_Bus** = 0	Liberar o barramento

Figura 131: Um fluxo de dados realizável pelo circuito da Figura 130

Exemplos de fluxos de dados realizáveis com o circuito da Figura 130 são mostrados na Figura 131 e na Figura 132.

Copiar o conteúdo da posição 15 da memória para o registrador Out		
Passo	Sinal	Comentário
1	**Input** = 15	Colocar 15 (em binário, 1111) na entrada **Input**
2	**In_Clk** = 1	Copiar no registrador **In** a entrada **Input**
3	**In_Clk** = 0	Zerar o clock do registrador **In**
4	**In_Bus** = 1	Colocar o valor de **In** no barramento
5	**MAR_Clk** = 1	Copiar o valor do barramento no registrador **MAR** (e também na entrada **A** da RAM)
6	**MAR_Clk** = 0	Zerar o clock do registrador **MAR**
7	**In_Bus** = 0	Liberar o barramento
8	**RAM_Bus** = 1	Colocar o valor da posição **A** da memória no barramento
9	**Out_Clk** = 1	Copiar o valor do barramento no registrador **Out**
10	**Out_Clk** = 0	Zerar valor do clock do registrador **Out**
11	**RAM_Bus** = 0	Liberar o barramento

Figura 132: Outro exemplo de fluxo de dados realizável com o circuito da Figura 130

3.4 Acumuladores e Loops

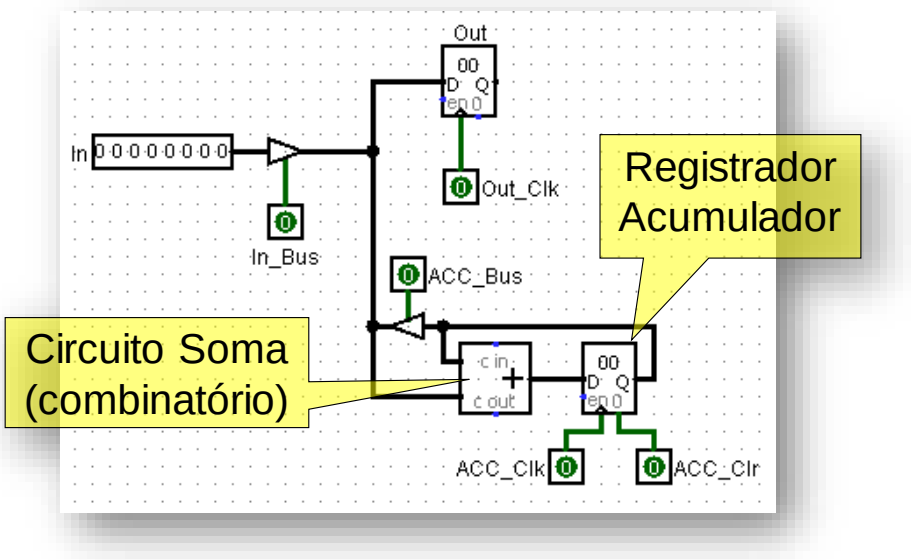


Figura 133: Um registrador acumulador

Um registrador pode ser usado como um *acumulador*, como mostrado na Figura 133. Neste arranjo,

- a entrada do acumulador é alimentada por um circuito combinatório que tipicamente realiza operações aritméticas ou lógicas, e
- a saída do acumulador realimenta o circuito combinatório, isto é, a saída do acumulador é uma das entradas do circuito combinatório.

Este arranjo permite, por exemplo, somar 10 números, seqüenciando as somas ao longo do tempo. Permitiria também, a uma pessoa que vivesse o tempo suficiente, somar um milhão de parcelas.

Calcular 2+7+4 e colocar o resultado no registrador Out		
Passo	Sinal	Comentário
1	ACC_Clr = 1	Zerar o acumulador
2	ACC_Clr = 0	
3	In = 2	Colocar 2 (em binário, 10) na entrada In
4	In_Bus = 1	Colocar o valor de In no barramento (a outra entrada do somador tem o valor do acumulador, que é 0)
5	ACC_Clk = 1	Copiar o valor da saída do somador no acumulador
6	ACC_Clk = 0	Zerar o clock do acumulador
7	In = 7	Colocar 7 (em binário, 111) na entrada In (e também no barramento). A saída do somador contém a soma do valor no barramento com o valor no acumulador
8	ACC_Clk = 1	Copiar o valor da saída do somador no acumulador
9	ACC_Clk = 0	Zerar o clock do acumulador
10	In = 4	Colocar 4 (em binário, 100) na entrada In
11	ACC_Clk = 1	Copiar o valor da saída do somador no acumulador
12	ACC_Clk = 0	Zerar o clock do acumulador
13	In_Bus = 0	Liberar o barramento
14	ACC_Bus = 1	Colocar o valor do acumulador no barramento
15	Out_Clk = 1	Copiar o valor do barramento no registrador Out
16	Out_Clk = 0	Zerar o clock do registrador Out
17	ACC_Bus = 0	Liberar o barramento

Figura 134: Exemplo de fluxo de dados realizável pelo circuito da Figura 133

A Figura 134 mostra um exemplo de uso de um acumulador para a soma de três parcelas.

3.5 Uma Calculadora

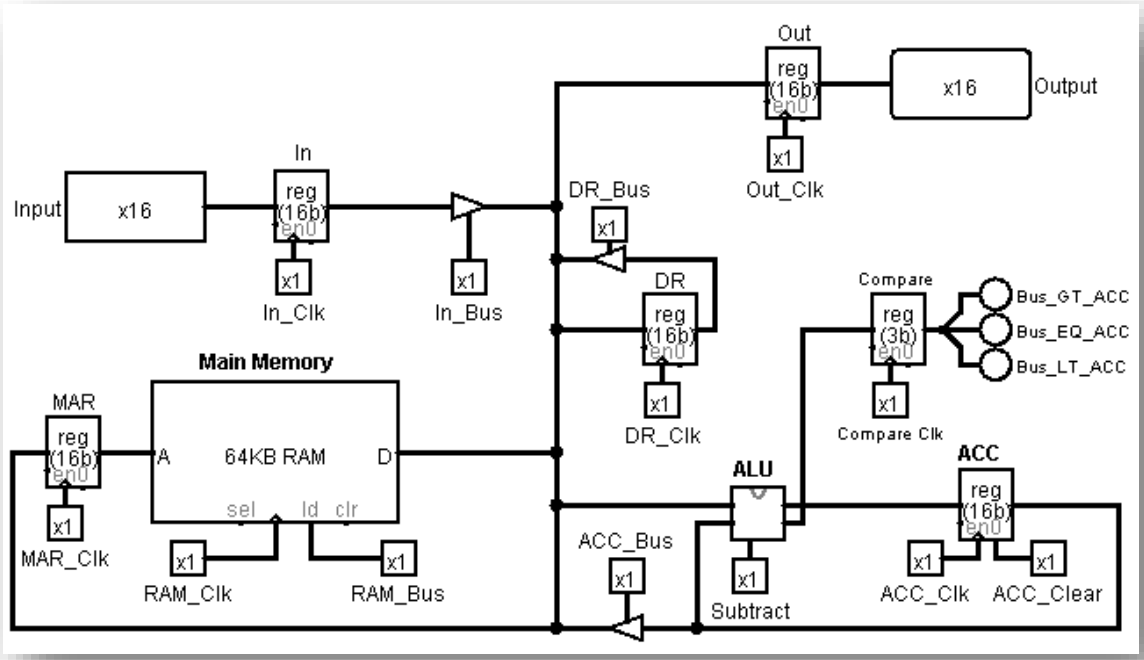


Figura 135: Uma calculadora

A Figura 135 mostra um circuito – uma calculadora – com diversos componentes ligados a um barramento de 16 bits:

- Registradores **In** e **Out**, ligados também a entradas e saídas de 16 bits
- Um registrador de dados, o **DR**;
- Uma memória principal com 32K palavras de 16 bits, também com 16 bits de endereço;
- Uma *unidade lógico-aritmética*, que é um circuito combinatório com duas entradas, uma ligada ao barramento e outra à saída do acumulador. A **ALU** (*Arithmetic and Logical Unit*) é capaz de realizar somas, subtrações e comparações entre suas duas entradas. (Outras operações que uma **ALU** poderia fazer incluem operações lógicas **AND**, **OR** e **NOT** e de deslocamento (*shift*) de bits.)
- Um registrador acumulador **ACC**, alimentado pela saída de resultado de operação da unidade lógico-aritmética;
- Um registrador de resultado de comparação **Compare**, também alimentado pela **ALU**. Este registrador armazena 3 bits, e sua saída está ligada a leds com os rótulos Bus_GT_ACC, Bus_EQ_ACC e Bus_LT_ACC. GT vem de *Greater Than*, EQ de *EQual*, e LT de *Less Than*.

Temos ainda o registrador **MAR**, que alimenta a entrada de endereço da memória principal, e que é alimentado pelo barramento. A biblioteca Entrada/Saída do Logisim oferece "leds", pequenas lâmpadas que foram acrescentadas ao circuito somente para acompanhamento visual do registrador **Compare**. Cada registrador, assim como a memória principal, tem um sinal de clock; cada saída para o barramento tem um sinal que controla a posse do barramento.

As rotas de dados de uma calculadora como a da Figura 135 permitem controlar diversos fluxos de dados – diversas computações – envolvendo a memória RAM, as entradas e saídas, o acumulador e os registradores de dados e de endereços. O controle do fluxo de dados é feito pelo usuário Logisim, que se encarrega de:

- mudar de 0 para 1 ou de 1 para 0 os sinais de controle de posse de barramentos e de cópia de registradores, na seqüência adequada ao efeito desejado, e de
- fornecer valores através do registrador **Input**.

Vamos usar a calculadora para resolver um problema simples de transformação de informação: queremos somar os conteúdos das posições 1 e 2 da memória, e colocar o resultado na posição 3. Podemos fazer isso através das etapas:

1. Carregar no acumulador o conteúdo da posição 1 da RAM
2. Somar ao acumulador o conteúdo da posição 2 da RAM
3. Armazenar o conteúdo do acumulador na posição 3 da RAM.

Na Figura 136 e na Figura 137 estão mostradas tabelas com uma sequência de ações a serem executadas por um operador Logisim. Essas ações são sinais de controle a serem aplicados à calculadora ou, em destaque, entradas de valores literais necessários para essa computação. Na notação utilizada, **ACC_Clear**=1, por exemplo, significa fazer a entrada **ACC_Clear** igual a 1; **Input**=3 significa colocar 3 (em binário, 11) na entrada **Input**. Essas tabelas de ações funcionam como um *programa*, a ser executado por uma pessoa utilizando a calculadora.

Sinal	Comentários
ACC_Clear = 1	Carrega no acumulador o conteúdo da posição 1 da RAM
ACC_Clear = 0	
Input = 1	
In_Clk = 1	
In_Clk = 0	
In_Bus = 1	
MAR_Clk = 1	
MAR_Clk = 0	
In_Bus = 0	
RAM_Bus = 1	
ACC_Clk = 1	
ACC_Clk = 0	
RAM_Bus = 0	

Input = 2	Soma ao acumulador o conteúdo da posição 2 da RAM
In_Clk = 1	
In Clk = 0	
In_Bus = 1	
MAR_Clk = 1	
MAR_Clk = 0	
In_Bus = 0	
RAM_Bus = 1	
ACC_Clk = 1	
ACC_Clk = 0	
RAM_Bus = 0	

Figura 136: Sinais de controle e, em destaque, entrada dos valores literais 1 e 2 para as etapas 1 e 2

Input = 3	Armazena o conteúdo do acumulador na posição 3 da RAM
In_Clk = 1	
In_Clk = 0	
In_Bus = 1	
MAR_Clk = 1	
MAR_Clk = 0	
In_Bus = 0	
ACC_Bus = 1	
RAM_Clk = 1	
RAM_Clk = 0	
ACC_Bus = 0	

Figura 137: Sinais de controle e entrada de valor literal 3 para a etapa 3

O momento de escrita do programa é diferente do momento de sua execução, e podemos distinguir entre valores que são conhecidos no momento da programação, como os endereços 1, 2 e 3, e valores que só serão conhecidos no momento da execução do programa, como os conteúdos das posições 1 e 2 da memória. Nós chamamos valores já conhecidos no momento da programação de *valores literais*, ou simplesmente *literais*.

3.6 Clocks

Nós vamos agora introduzir um novo tipo de circuito digital, completando os ingredientes que, como veremos no próximo módulo, nos permitirão transformar esta calculadora em um processador, isto é, em um circuito digital capaz de, automaticamente, executar as operações prescritas em um programa.

O ponto a resolver é o clicar automatizado de botões, isto é, a emissão de sinais de controle seqüenciados no tempo sem intervenção humana; para isso vamos precisar de um novo tipo de circuito. O motor, o propulsor de qualquer circuito digital é um *oscilador*, ou *clock*. Um clock é um dispositivo cuja saída oscila entre 0 e 1 em uma freqüência conhecida. Um computador de 1 GHz (1 Giga Hertz) utiliza um clock cuja saída varia entre 0 e 1, um bilhão de vezes por segundo. O Logisim oferece clocks simulados, para os quais o usuário pode escolher a freqüência de oscilação, como mostra a Figura 138.

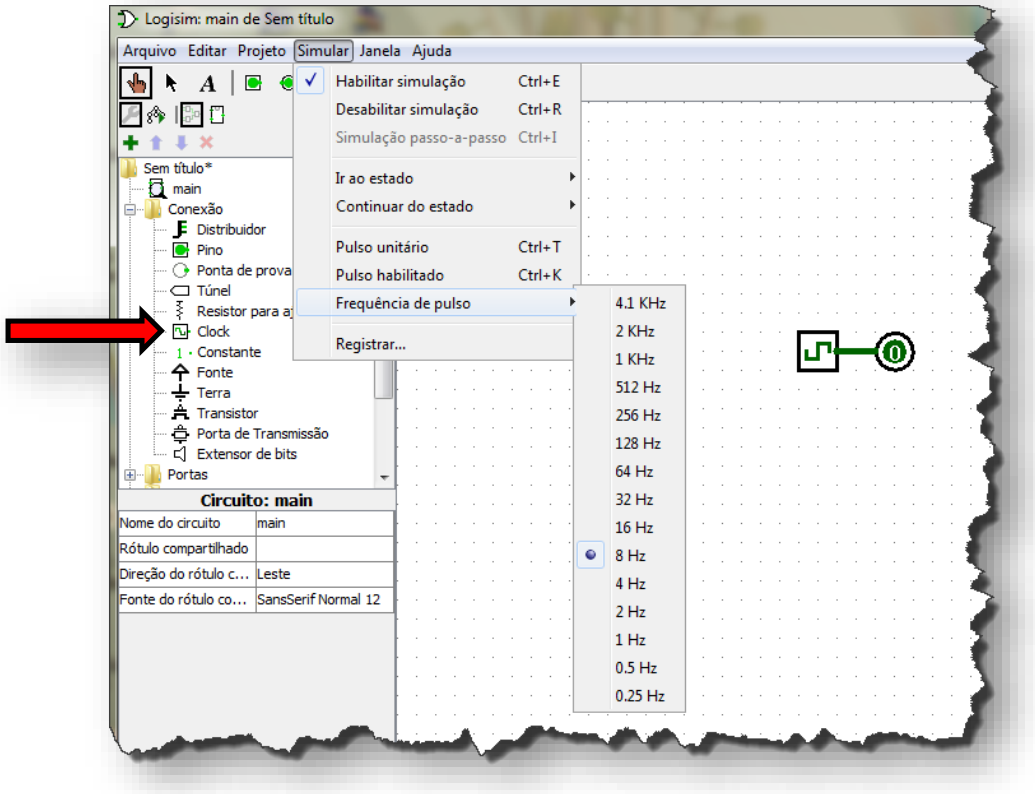


Figura 138: Um clock no Logisim, com o menu de escolha de freqüência

A partir do sinal básico fornecido por um clock, circuitos como *registradores circulares* podem fornecer outros sinais, que podem ser usados para coordenar ao longo do tempo o fluxo de dados em um circuito.

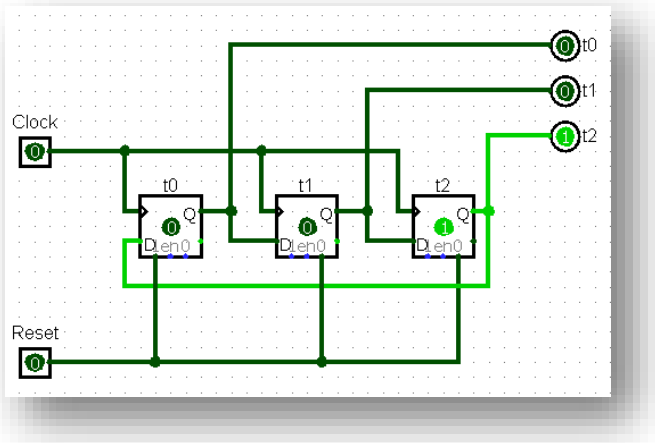


Figura 139: Um registrador circular

A Figura 139 mostra um registrador circular formado por três flip-flops tipo D alimentados por um clock. O cabeamento é tal que a saída do FF **t0** está ligada à entrada **D** do FF **t1**, a saída do FF **t1** à entrada do FF **t2**, e a saída do FF **t2** está ligada à entrada do FF **t0**, em um arranjo circular. O registrador é inicializado através do pino **Reset**, que coloca 1 no flip-flop **t0**, e 0 nos demais. A cada subida do clock cada FF copia a sua entrada, o que faz com que o 1 inicialmente armazenado no FF **t0** passe para o FF **t1**, depois para o **t2**, retornando então ao FF **t0**.

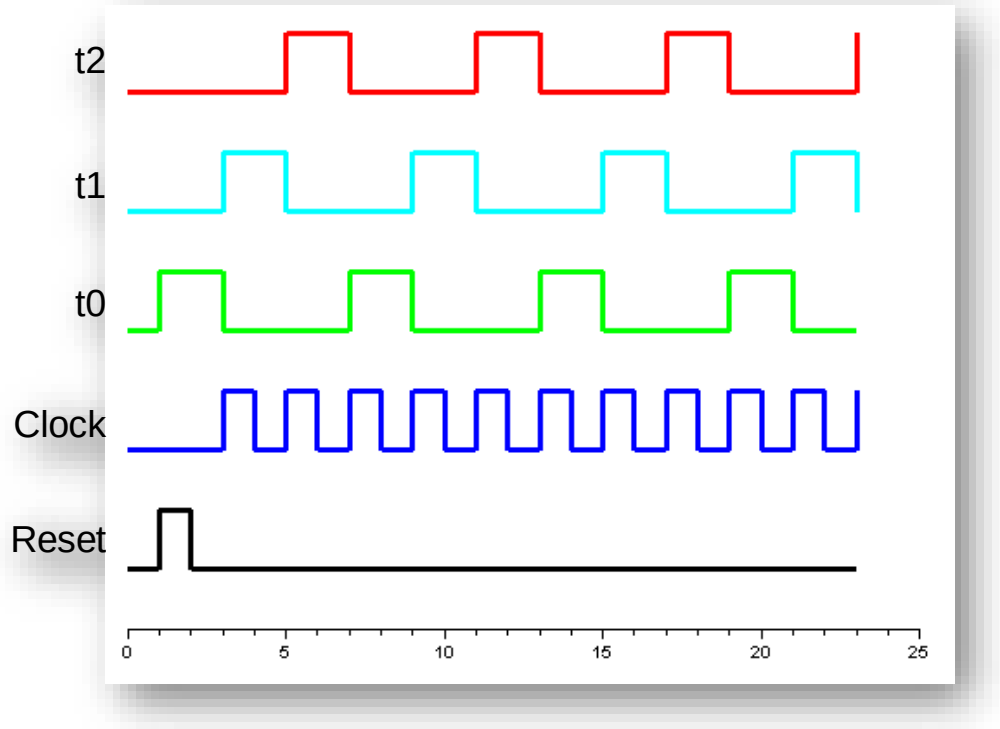


Figura 140: Carta de tempo para o registrador circular da Figura 139

A Figura 140 mostra a evolução temporal de um registrador circular. É importante observar que um registrador circular pode ter quantos flip-flops se queira e que, com isso, conseguimos obter sinais distribuídos no tempo na forma como desejarmos.

Temos agora todos os ingredientes necessários para a construção de um processador, o que faremos no próximo módulo.

4 Processadores

4.1 Programa Armazenado

Um processador é um circuito digital com comportamento flexível, comandado por uma coisa que chamamos de *programa*. Um programa é produzido por um ser humano, que deseja resolver um problema de transformação de informação. Trocando-se o programa, troca-se o comportamento do processador. Isso não deve envolver modificações no circuito, pois a idéia é que essa troca de programa seja uma operação de fácil realização.

Se é isso o que queremos, um programa só pode ser, em seu formato final, informação codificada em bits, que deve ser carregada em alguma memória para sua execução. Para eliminar a necessidade de intervenção humana durante a execução do programa, uma *unidade de controle* deve ser adicionada à calculadora. Ao executar um programa, a unidade de controle deve se encarregar das tarefas executadas na calculadora pelo operador Logisim, que são:

- emitir sinais de controle, e
- fornecer valores literais.

Vamos construir uma primeira unidade de controle usando as seguintes idéias:

- para sua execução, o programa deve ficar armazenado como uma seqüência de palavras em uma memória RAM; por razões que veremos depois, chamamos cada palavra desses programas de *micro-instrução*;
- a cada sinal de controle da calculadora deve corresponder um bit nas micro-instruções;
- a unidade de controle implementa um ciclo de leitura em seqüência de micro-instruções da memória com o programa, manipulando adequadamente a entrada de endereço da memória;
- em cada ciclo, os bits de cada palavra lida são encaminhados para as saídas da unidade de controle, que são ligadas aos pontos de controle da (ex-) calculadora.

Endereço	Micro-instrução 23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Clicks Equivalentes	Efeito
	Sinais											n_bus	n_Clk	MAR_Clk	AM_Bus	AM_Clk	R_Bus	R_Clk	ubtract	ut_Clk	ompare_Clk	CC_Bus	CC_Clk	CC_Clear		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	CC_Clear = 1	ACC = 0
1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	CC_Clear = 0; In_Bus = 1	Bus = In
2	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	AR_Clk = 1	MAR = Bus
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	n_Bus = 0; MAR_Clk = 0; DR_Bus = 1	Bus = DR
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	AM_Clk = 1	RAM = Bus

Micro-instruções

Figura 141: Codificação de sinais de controle em micro-instruções

A Figura 141 mostra como podemos especificar micro-instruções por meio de uma tabela onde cada coluna da parte mais à esquerda corresponde a um dos sinais de controle do circuito da

Figura 135. Cada linha da tabela corresponde a uma micro-instrução, e as micro-instruções serão executadas sequencialmente pela unidade de controle. As duas colunas mais à direita têm informações para consumo humano: os clicks que o operador Logisim deveria realizar, e o efeito sobre o estado do circuito. Repare que desta forma é possível ligar ou desligar mais de um botão ao mesmo tempo.

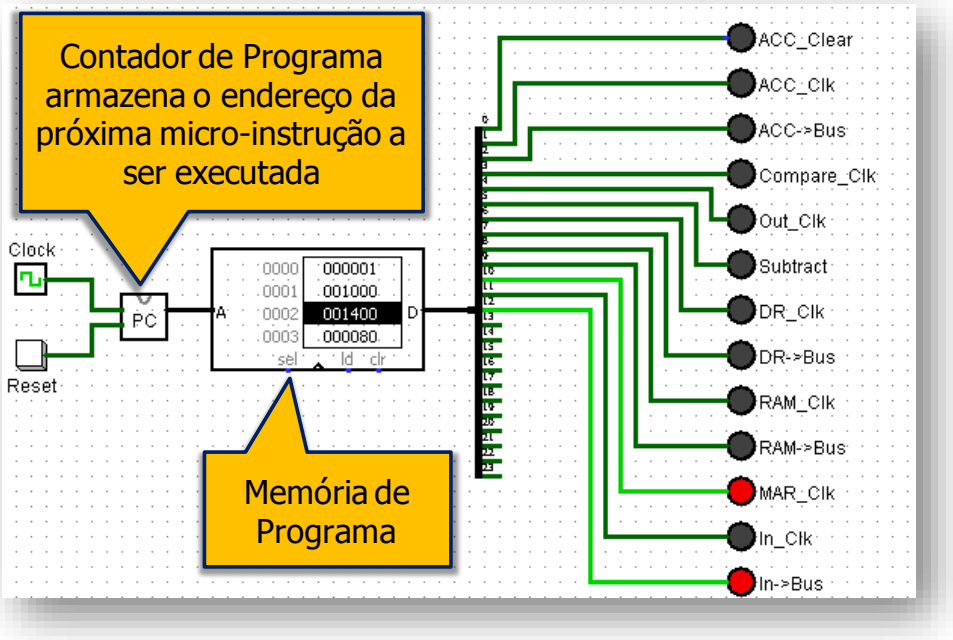


Figura 142: Circuito para geração automática de sinais de controle segundo um programa armazenado em uma memória RAM

Nós vamos agora mostrar um circuito que gera automaticamente sinais de controle, ignorando por enquanto o problema do fornecimento de literais. No circuito da Figura 142 nós vemos:

- uma memória RAM onde fica armazenado um programa (nós veremos mais tarde como colocar um programa na memória),
- saídas de sinais de controle, ligadas diretamente à saída da memória de programa, e
- um *contador de programa*, denominação que se dá a registradores com a função de controlar o endereço aplicado a memórias de programa, e que contêm ou o endereço da instrução em execução, ou o endereço da próxima instrução a ser executada.

Por uma questão de uniformidade com os outros processadores que examinaremos aqui, a memória de programa tem palavras de 24 bits, o que faz com que os bits de 12 a 23 não estejam sendo utilizados. O contador de programa **PC** (*Program Counter*) tem o seu valor incrementado de 1 a cada subida do clock, o que faz com que a saída de dados da memória exponha em sucessão as micro-instruções. Como cada micro-instrução determina o valor dos sinais de saída, nós temos o que queríamos: a geração automática de sinais de controle, guiada por um programa carregado na memória.

Figura 143: Uma caixa de música [38]; o cilindro tem o papel da memória com um programa; as saliências no cilindro têm papel análogo ao dos bits ligados em uma micro-instrução

O comportamento deste circuito lembra o de uma caixa de música como a que mostra a Figura 143 (não deixe de ver também o vídeo no YouTube; é muito bom), onde o cilindro tem o papel de uma memória onde está gravado um programa..

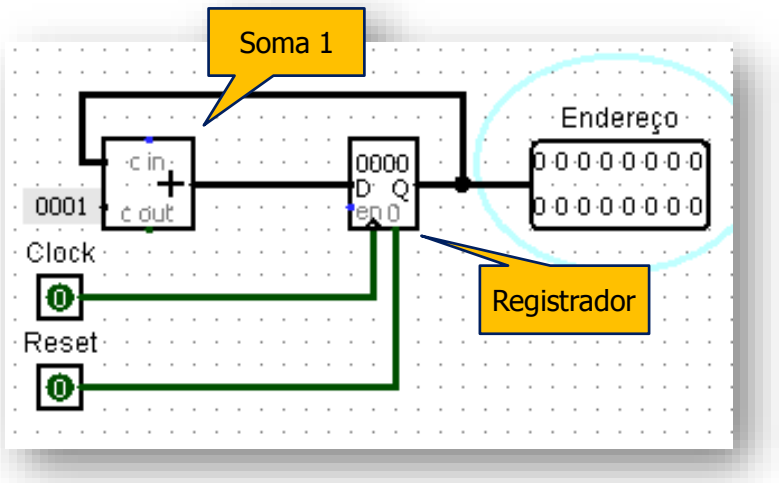


Figura 144: Um contador de programa simples

A Figura 144 mostra o circuito do contador de programa, que é um registrador cuja saída é uma das parcelas de um circuito de soma , cuja saída realimenta a entrada do registrador. A outra parcela da soma é sempre igual a 1, o que produz o efeito que desejamos para a execução das micro-instruções em sequëncia: a cada subida do clock, o valor do **PC** é incrementado de 1. O circuito possui também uma entrada **Reset** que zera o **PC**.

[38] bigbluesky2002, "YouTube - Swiss Antique Music Box", acessado 1 de abril de 2011, http://www.youtube.com/watch?v=-tzWt1X3oOg&p=66A384A130B0DCE4.

4.2 CPU-0: Um Primeiro Processador

Dominado o problema da emissão dos sinais de controle, vamos agora ver como eliminar a necessidade de intervenção do operador Logisim também na entrada de valores literais (endereços da memória, valores a serem adicionados ou carregados no acumulador, etc.). Nós queremos agora permitir que literais possam ser especificados no programa, e que seus valores sejam fornecidos pela unidade de controle à (ex-) calculadora nos momentos adequados.

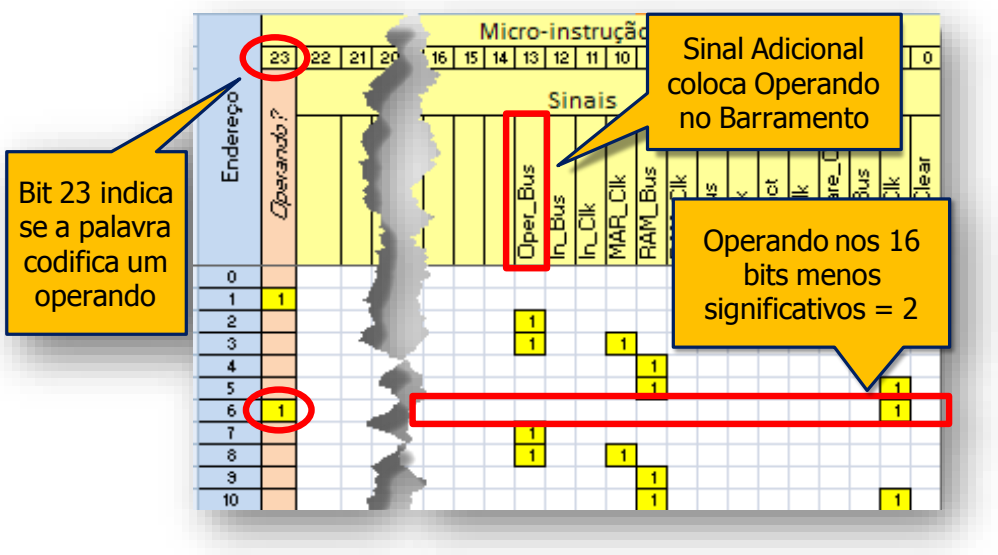


Figura 145: Codificação de operandos em micro-instruções

A Figura 145 mostra uma forma de se codificar valores literais em micro-instruções. Por motivos de ordem prática, nós adotamos micro-instruções de 24 bits. Para indicar que uma micro-instrução codifica um valor literal, nós vamos utilizar o bit 23, o mais significativo. Se este bit for igual a 0, os bits restantes são interpretados como sinais de controle; se for igual a 1, os 16 bits menos significativos são a codificação em binário de um valor literal (o barramento da calculadora tem 16 bits de largura).

A unidade de controle deverá ter uma saída com o valor do literal, ligada ao barramento da calculadora, e utilizando, como todas as outras ligações de saída para o barramento, um buffer controlado para evitar conflitos. Este buffer controlado é comandado por um sinal, que deve ser adicionado aos sinais já emitidos pela unidade de controle.

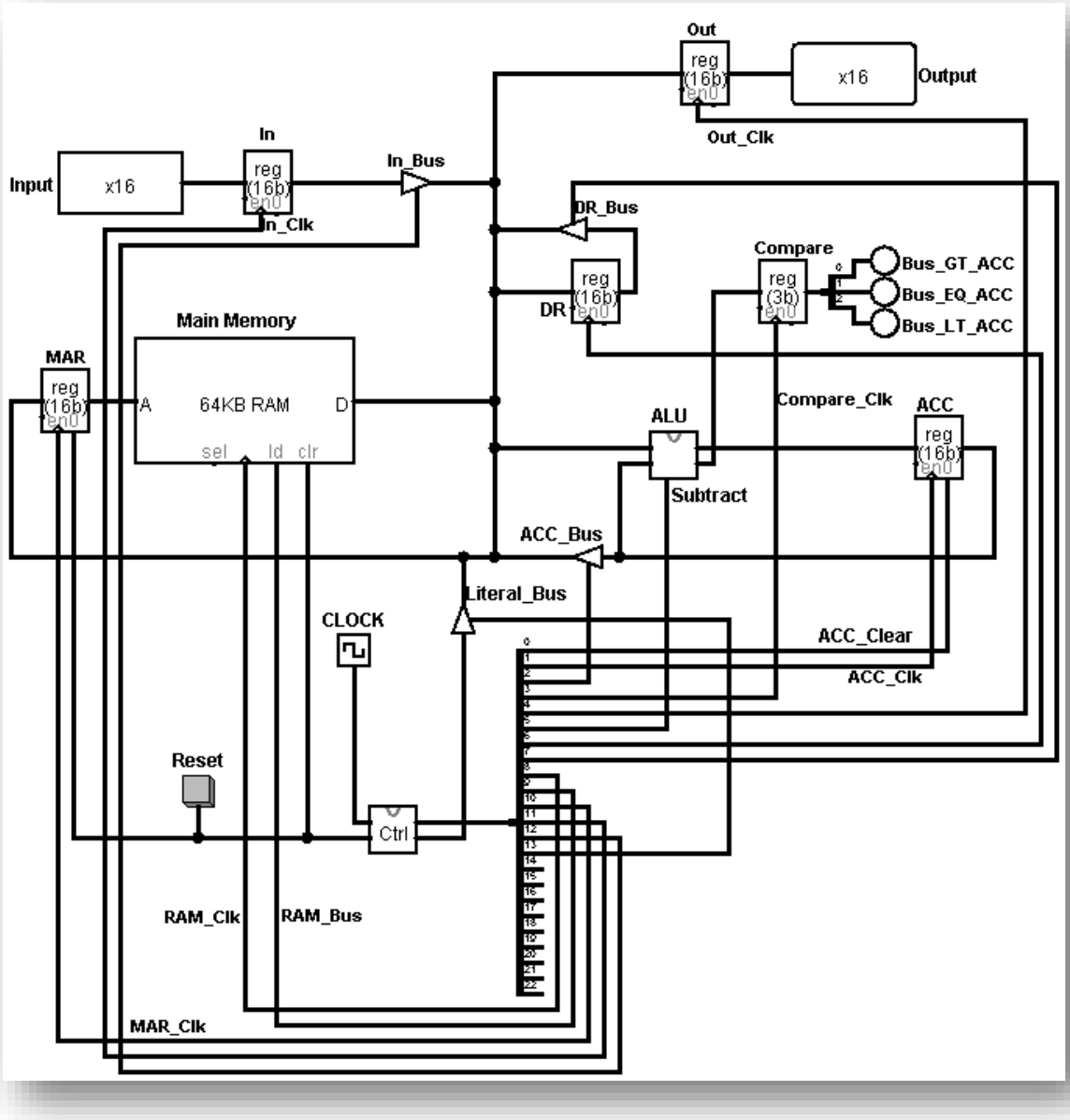


Figura 146: CPU-0: nosso primeiro processador

Já temos agora condições de mostrar o nosso primeiro processador, capaz de executar micro-instruções em sequência, com sinais de controle e valores literais fornecidos por uma unidade de controle. A Figura 146 mostra o circuito principal da **CPU-0**, nome que demos a este processador. Para compreendê-lo, você deve primeiramente reparar que podemos dividi-lo em duas partes. Na metade superior você deve reconhecer a nossa calculadora, onde todos os sinais de controle foram ligados à saída da unidade de controle (o bloquinho escrito **Ctrl**), que fica na metade inferior do circuito.

Você deve ainda reparar que a unidade de controle também tem uma saída ligada ao barramento da calculadora; é por esta saída que passam os literais especificados nas micro-instruções.

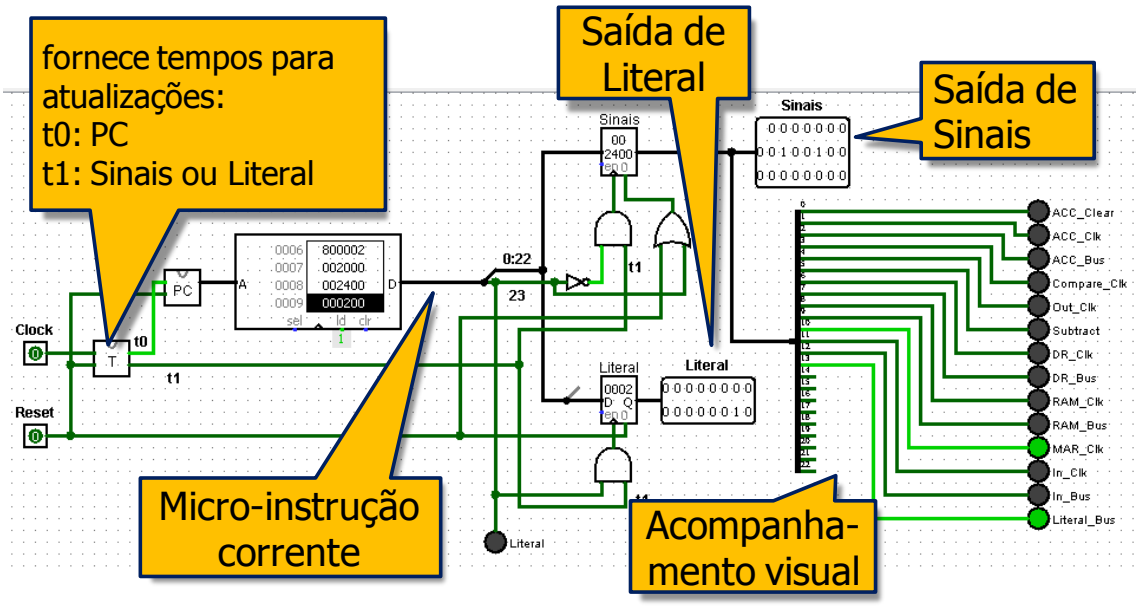


Figura 147: Unidade de controle da CPU-0

A Figura 147 mostra os detalhes internos da unidade de controle da **CPU-0**. Você deve reparar que a unidade de controle possui:

- registradores de sinais e de literal, que armazenam os bits fornecidos pelas saídas de mesmo nome;
- uma memória que armazena o programa; a micro-instrução corrente é o conteúdo da posição de endereço **PC**, que é colocada na saída **D** da memória;
- um circuito de temporização **T**, similar ao que já vimos na Figura 139, e que ao longo do tempo fornece os sinais **t0** e **t1** conforme mostrado na Figura 140;
- alguns LEDs sem participação na lógica do circuito, mas que ajudam no acompanhamento visual da simulação.

É através dos sinais fornecidos pelo circuito de temporização que o ciclo de execução de micro-instruções é implantado na **CPU-0**:

- o sinal **t0** atualiza o valor do **PC**;
- o sinal **t1** faz com que ou o registrador **Sinais**, ou o registrador **Oper**, copie sua entrada, com a escolha dentre estes dois sendo determinada pelo bit 23 da micro-instrução corrente.

Muito bem, já temos um circuito que executa programas formados por micro-instruções, onde cada micro-instrução codifica sinais de controle ou literais, sem necessidade de intervenção humana na execução. Mas temos ainda que resolver dois problemas: como construir um programa, e como fazer para colocar este programa na memória de micro-instruções da **CPU-0**.

Endereço	Micro-instrução																					Efeito	
	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	Sinais																									
	Literal?										Literal_Bus	In_Dbus	In_Clk	MAR_Clk	RAM_Bus	RAM_Clk	DR_Bus	DR_Clk	Subtract	Out_Clk	Compare_Clk	ACC_Bus	ACC_Clk	ACC_Clear	Da micro-instrução	Acumulado
0																								1	ACC = 0	Carrega no acumulador o conteúdo da posição 1 da RAM
1	1																							1	Literal = 1	
2											1														Bus = Literal	
3											1			1											MAR = Bus	
4															1										Bus = RAM	
5															1								1		ACC = ACC + Bus	
6	1																						1		Literal = 2	Soma ao acumulador o conteúdo da posição 2 da RAM
7											1														Bus = Literal	
8											1			1											MAR = Bus	
9															1										Bus = RAM	
10															1								1		ACC = ACC + Bus	
11	1																						1	1	Literal = 3	Armazena o conteúdo do acumulador na posição 3 da RAM
12											1														Bus = Literal	
13											1			1											MAR = Bus	
14																						1			Bus = ACC	
15																1						1			RAM = Bus	

Figura 148: Construindo um programa para a CPU-0

Para programar a CPU-0 nós podemos usar uma tabela como a da Figura 148, cujas colunas se dividem em 3 agrupamentos:

- *Endereço*, que indica em qual posição da memória de programa a micro-instrução deve ser armazenada,
- *Micro-instrução*, onde são colocados pelo programador os 24 bits que compõem a micro-instrução propriamente dita (na figura posições com valor zero estão em branco, para não poluir visualmente a tabela), e
- *Efeito*, que contém informações para consumo humano, indicando tanto o efeito de cada micro-instrução como o efeito acumulado de grupos de micro-instruções.

Os bits marcados no agrupamento de micro-instruções constituem uma imagem que deve ser carregada na memória de programa da CPU-0. O programa da Figura 148 corresponde exatamente aos sinais de controle e entradas de operandos mostrados na Figura 136 e na Figura 137; sua execução tem portanto o mesmo efeito: somar os conteúdos das posições 1 e 2 da memória, e colocar o resultado na posição 3.

Construída a tabela-programa, os bits das micro-instruções devem ser armazenados em alguma mídia, e carregados na memória de programa do processador. Nos computadores atuais a carga de programas é feita por um outro programa, chamado de carregador ou *loader*.

Sim, mas como é que um loader vai parar na memória do computador? Nos computadores atuais, um loader primitivo é gravado pelo fabricante em uma memória ROM (*Read Only Memory*), e é executado no momento em que o computador é ligado, constituindo a primeira etapa de um procedimento que tem o nome de *bootstrap*. Usando um disco magnético (tipicamente), o loader primitivo carrega um outro loader, mais sofisticado, que por sua vez carrega outro mais sofisticado ainda, e isso termina com a carga do sistema operacional. O uso normal de programas utiliza um loader do sistema operacional.

Figura 149: Painel de um PDP11/70, de 1975[39]

Nem sempre foi assim. Quando ainda não existiam memórias ROM, o loader primitivo era carregado palavra por palavra, através do painel do computador. Como você pode ver na Figura 149, o painel continha uma série de chaves cujo posicionamento ligava ou desligava um bit, e botões para carregar o registrador de endereço ou o conteúdo de uma posição de memória com os bits definidos pelas chaves. Não era necessário fazer isso a cada vez que se ligava o computador: a memória principal daquele tempo utilizava núcleos de ferrite, e não era volátil. Reservavam-se algumas posições de memória para conter o loader, e a carga através do painel só era necessária quando, por um erro de programação, alguém escrevia sobre essas posições da memória.

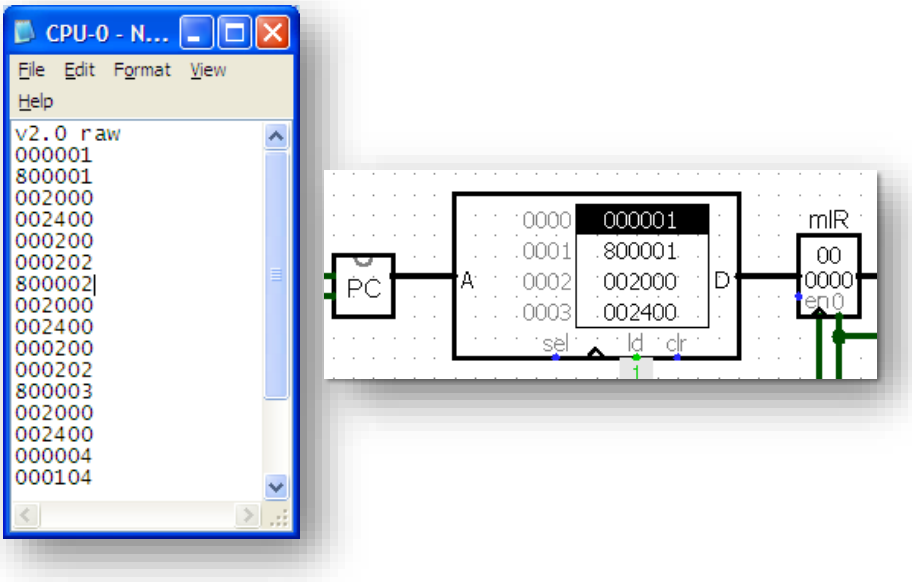


Figura 150: Arquivo com mapa de memória usado pelo Logisim

Aqui nós não trabalhamos com memórias reais, e sim com memórias simuladas pelo Logisim. Isso nos permite escrever diretamente valores para posições de memória, ou ler de um

[39] "File:Pdp-11-70-panel.jpg - Wikipedia, the free encyclopedia", acessado 2 de setembro de 2011, http://en.wikipedia.org/wiki/File:Pdp-11-70-panel.jpg.

arquivo um mapa da memória (para ter acesso a essas operações, clique com o botão direito do mouse sobre a memória). A Figura 150 mostra o formato de um arquivo com a codificação em hexadecimal do programa da Figura 148 e a memória de programa Logisim após a carga deste arquivo. No site do curso você irá encontrar planilhas que auxiliam na produção de arquivos-programas em hexadecimal.

4.3 CPU-1: Processador com Instruções de Desvio

Suponha agora que queremos construir para a **CPU-0** um programa que some os conteúdos das posições 1, 2, 3, 4 e 5 da memória principal, e coloque o resultado na posição 6. Não é difícil: basta acrescentar ao programa mais passos de somas ao acumulador, como ilustra a Figura 151.

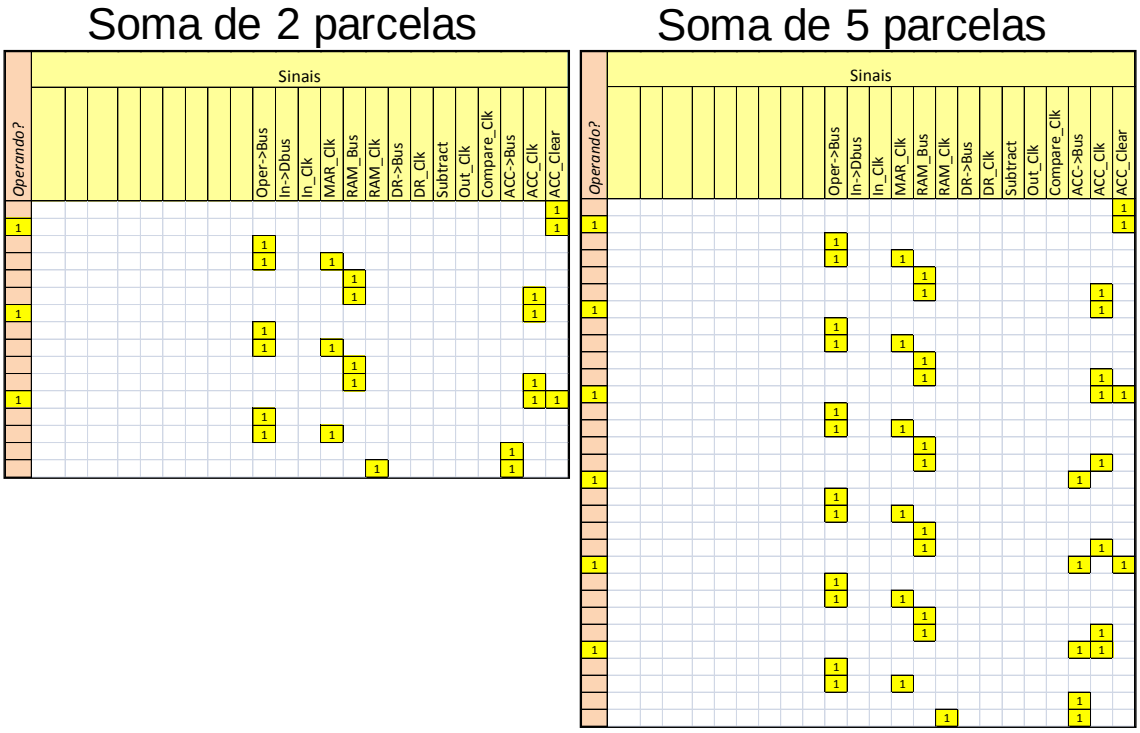


Figura 151: Programas para a CPU-0 que somam 2 e 5 parcelas

Nós sabíamos que era possível realizar computações arbitrariamente complicadas com a nossa calculadora, aplicando manualmente sinais de controle, e entrando também manualmente com os operandos necessários. Com a **CPU-0**, nós mostramos que é possível automatizar essas operações, com o uso de uma unidade de controle impulsionada por um clock. Mas o exemplo da soma de 5 parcelas nos mostra um problema: na **CPU-0**, um programa cresce de tamanho com o número de operações que realiza. Qual seria o tamanho de um programa que soma um milhão de parcelas?

Para conseguir escrever programas cujo tamanho não cresça com o número de operações que sua execução realiza, precisamos alterar nosso modelo de execução seqüencial de micro-instruções. Nós vamos agora apresentar um outro processador, a **CPU-1**, que possui uma micro-instrução especial que *desvia* o fluxo de execução para um endereço designado na memória de programa. O ciclo de leitura e execução de micro-instruções deve ser modificado em função disso, pois a próxima micro-instrução a ser executada nem sempre é a que está armazenada no endereço consecutivo da memória de programa.

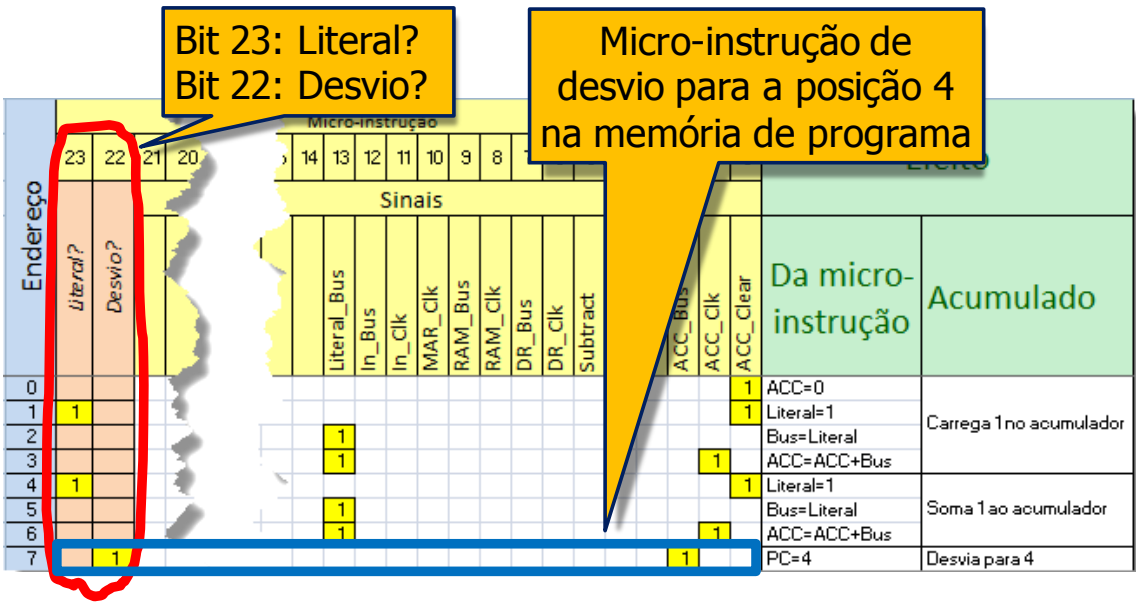


Figura 152: Formato de micro-instruções que contempla instruções de desvio

A Figura 152 mostra como as micro-instruções são codificadas na **CPU-1**:

- Se o bit 23 (o mais significativo) for igual a 1, a micro-instrução é de desvio. A próxima micro-instrução a ser executada é aquela armazenada no endereço codificado nos 16 bits menos significativos.
- Se o bit 22 for igual a 1, a micro-instrução contém um valor literal, codificado (como na CPU-0) nos 16 bits menos significativos.
- Se os bits 22 e 23 forem iguais a zero, temos uma micro-instrução de sinais;
- Os bits 22 e 23 nunca devem ser ambos iguais a 1 em uma micro-instrução.

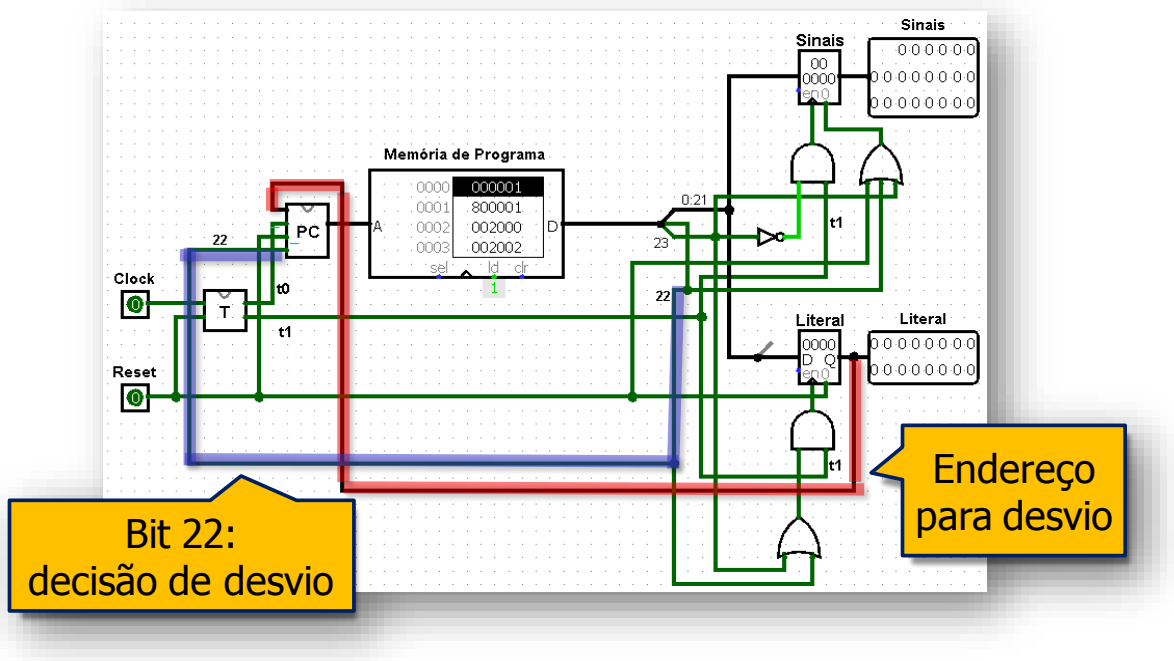


Figura 153: Unidade de controle da CPU-1, com destaque para a conexão do registrador de operando com o contador de programa e para a decisão de desvio, indicada pelo bit 22 da micro-instrução corrente

Para alterar o ciclo de micro-instrução nós temos que modificar a unidade de controle e o contador de programa da **CPU-0**. Na Figura 153 nós vemos a unidade de controle da **CPU-1**, onde você deve atentar para as seguintes modificações:

- O contador de programa tem duas entradas adicionais. Uma delas está conectada ao registrador **Oper**, e recebe o endereço para um possível desvio. A outra entrada está conectada ao bit 23 do registrador de micro-instrução que, como vimos, indica se a micro-instrução corrente é de desvio.
- O registrador **Oper** é usado para armazenar o endereço de desvio.
- O registrador **Sinais** é zerado se o bit 22 ou o bit 23 da micro-instrução corrente for igual a 0.

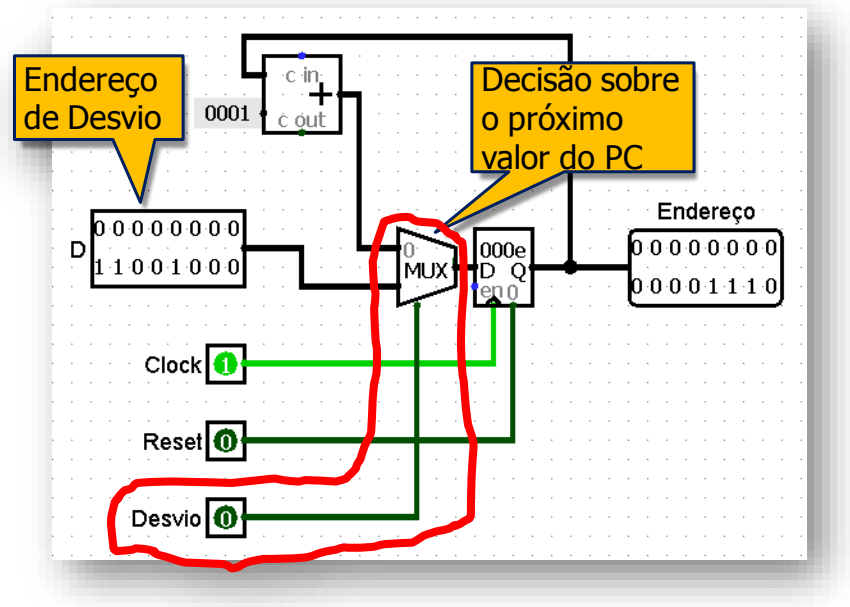


Figura 154: O contador de programa da CPU-1, com possibilidade de imposição de um novo endereço arbitrário

A Figura 154 mostra o novo contador de programa, onde você deve reparar que a entrada do registrador de endereço está conectada à saída de um multiplexador. Este multiplexador encaminha para a entrada ou bem o valor corrente do **PC** acrescido de 1 (fornecido pela saída do somador), quando a entrada **Desvio** é igual a 0, ou então o valor da entrada **D**, quando a entrada **Desvio** é igual a 1.

Endereço	Micro-instrução																					Efeito	
	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	Sinais																									
	Literal?	*Desvio?*									Literal_Bus	In_Bus	In_Clk	MAR_Clk	RAM_Bus	RAM_Clk	DR_Bus	DR_Clk	Subtract	Out_Clk	Compare_Clk	ACC_Bus	ACC_Clk	ACC_Clear	Da micro-instrução	Acumulado
0																								1	ACC=0	Carrega 1 no acumulador
1	1																							1	Literal=1	
2											1														Bus=Literal	
3											1												1		ACC=ACC+Bus	
4	1																							1	Literal=1	Soma 1 ao acumulador
5											1														Bus=Literal	
6											1												1		ACC=ACC+Bus	
7		1																				1			PC=4	Desvia para 4

Figura 155: Um programa para a CPU-1

Na Figura 155 nós vemos uma tabela com um programa para a **CPU-1**, cujo efeito é muito simples: o programa usa o acumulador para contar 1, 2, 3, ... indefinidamente. Para executar este programa no Logisim as etapas são as mesmas: as micro-instruções devem ser gravadas em um arquivo, codificadas em hexadecimal, uma em cada linha. Este arquivo deve ser carregado na memória de programa na unidade de controle da **CPU-1**.

4.4 CPU-2: Processador com Instruções de Desvio Condicional

Com a micro-instrução de desvio da **CPU-1** nós conseguimos construir programas que prescrevem a repetição de ações por um processador e, com isso, desvincular o tamanho de um programa do número de operações realizadas em sua execução. Este é um resultado muito importante, pois programas são feitos por nós, humanos, que queremos trabalhar pouco, e são executados por computadores, que não se importam de trabalhar muito.

Mas como fazer para interromper as repetições? Afinal, um loop precisa parar. Nós queremos poder construir programas que resolvam problemas como "somar os conteúdos das posições de memória com endereços entre 100 e 200", ou "encontrar o menor valor entre os conteúdos das posições de memória com endereços entre 1000 e 1.000.000", que certamente envolvem loops, mas que devem ser interrompidos ao se atingir os limites das operações desejadas.

Este problema é resolvido por micro-instruções de *desvio condicional*, que provocam desvios no fluxo de execução somente quando o resultado de comparações satisfizer uma condição (maior, igual, menor, maior ou igual, etc.).

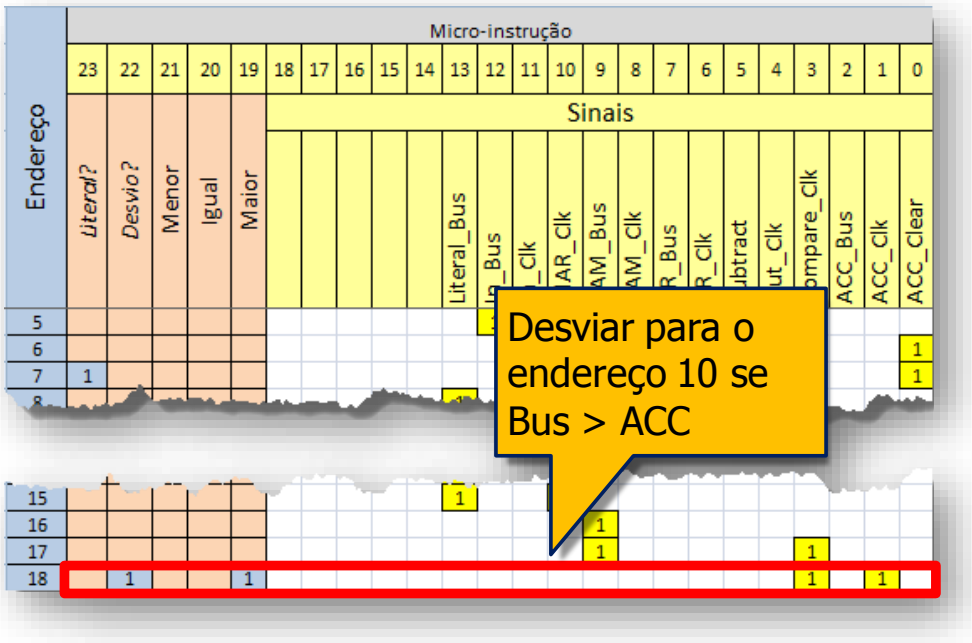


Figura 156: Codificação de micro-instruções de desvio condicional

Na Figura 156 você pode ver a codificação de micro-instruções que iremos adotar para um novo processador, a CPU-2. Nessa codificação,

- o bit 22 indica se a micro-instrução é de desvio;
- o bit 23, como na **CPU-1**, indica que a micro-instrução contém um valor literal;
- os bits 21, 20 e 19 são usados nas micro-instruções de desvio, e especificam as condições em que o desvio deve ser efetivamente realizado, em função do valor corrente do registrador de comparação. Se, por exemplo, tivermos uma micro-

instrução de desvio com os bits 21 e 20 iguais a 1, e o bit 19 igual a zero, na execução desta micro-instrução o desvio ocorrerá somente se o registrador de comparação estiver seja com a saída **D<ACC** ligada, seja com a saída **D=ACC** ligada. Um desvio incondicional pode ser obtido colocando estes 3 bits iguais a 1.

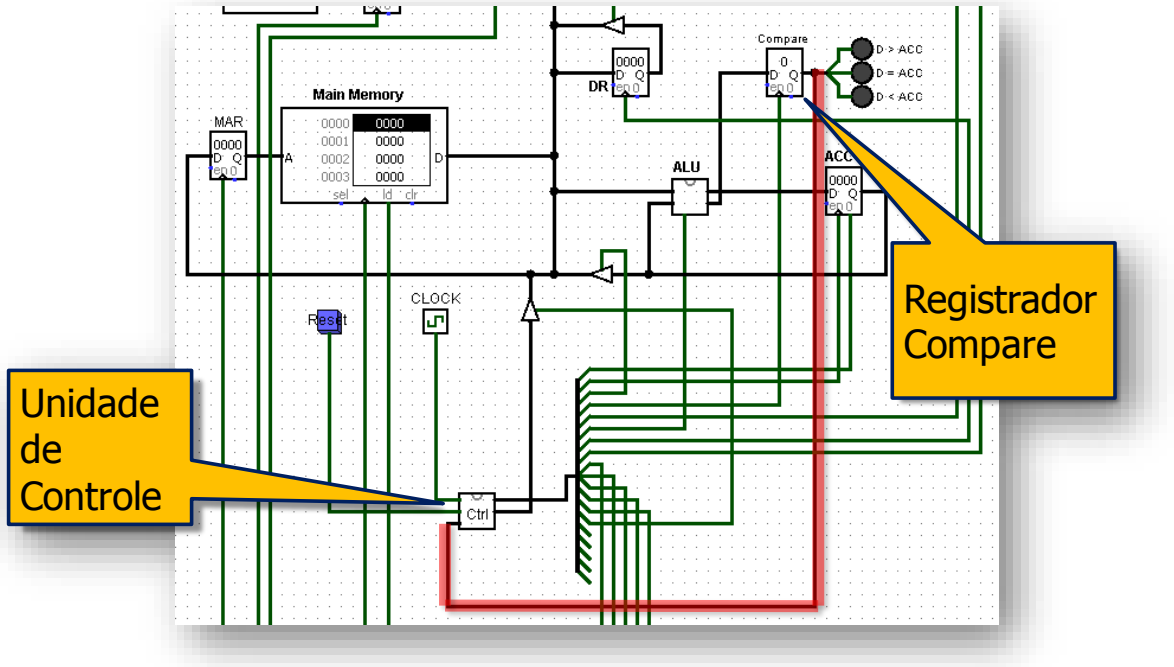


Figura 157: Parte da CPU-1, destacando a alimentação da saída do registrador de comparação como entrada adicional da unidade de controle

Na Figura 157 você pode ver que a saída do registrador de comparação alimenta agora a unidade de controle, fornecendo a informação necessária para as decisões de desvio condicional.

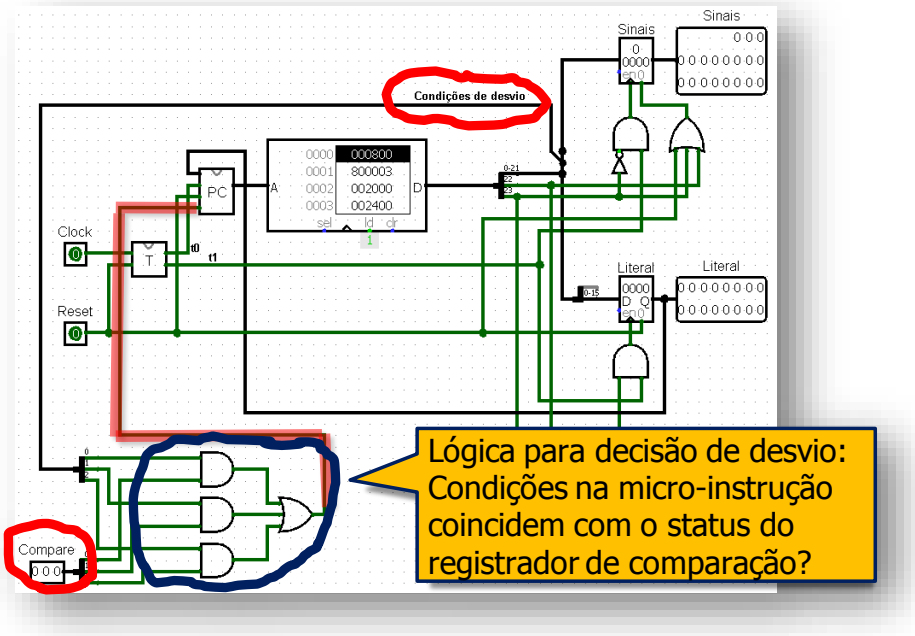


Figura 158: A unidade de controle da CPU-2, com destaque para a lógica de desvio

Quanto à unidade de controle, as novidades são (veja a Figura 158):

- temos uma entrada adicional que, como já dissemos, é alimentada pelos três bits do registrador de comparação da calculadora
- esses três bits que vêm do registrador de comparação são confrontados com os três bits (bits 21, 20 e 19) da micro-instrução que, conforme a Figura 156, especificam a condição de desvio . Essa confrontação é dada pela expressão booleana

$$\textbf{Desvio} = (\textbf{b21.D<ACC}) + (\textbf{b20.D=ACC}) + (\textbf{b19.D>ACC})$$

que coloca o valor 1 na entrada **Desvio** do **PC** (isto é, determina a realização do desvio) quando pelo menos uma das condições de desvio é atendida pelo estado do registrador de comparação.

Usando estas micro-instruções de desvio condicional, nós pudemos desenvolver o programa da Figura 159, que também irá usar o acumulador para contar 1, 2, 3, ..., mas que interrompe a contagem quando o valor do acumulador atingir um valor colocado antes do início da simulação na entrada **In**.

Endereço	Micro-instrução																					Efeito	
	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	Sinais																								Da micro-instrução	Acumulado
	Literal?	*Desvio?*	Menor	Igual	Maior						Literal_Bus	In_Bus	In_Clk	MAR_Clk	RAM_Bus	RAM_Clk	DR_Bus	DR_Clk	Subtract	Out_Clk	Compare_Clk	ACC_Bus	ACC_Clk	ACC_Clear		
5												1				1									RAM=Bus	
6																								1	ACC=0	Carrega 1 no acumulador
7	1																							1	Literal=1	
8											1														Bus=Literal	
9											1												1		ACC=ACC+Bus	
10	1																							1	Literal=1	Soma 1 ao acumulador
11											1														Bus=Literal	
12											1												1		ACC=ACC+Bus	
13	1																						1	1	Literal=3	Compara o conteúdo da posição 3 da memória com o acumulador
14											1														Bus=Literal	
15											1			1											MAR=Bus	
16															1										Bus=RAM	
17															1						1				Compare=Bus::ACC	
18		1			1																1		1		se Bus_GT_ACC, PC=10	Se Bus>ACC, desvio para 10

Figura 159: Um programa para a CPU-2

Este programa inicia armazenando o valor encontrado na entrada **Input** (e que deve ser colocado ali antes do início da simulação) na posição 3 da memória. Em seguida acumulador é inicializado com o valor 1. Segue-se um loop de soma e de comparação, que inclui uma micro-instrução de desvio condicional.

4.5 CPUs Reais: Instruções e Programação em Assembler

Com a **CPU-2** nós conseguimos construir programas que prescrevem operações repetitivas para execução por um processador, e conseguimos também, com desvios condicionais, interromper em momentos adequados essas repetições. Isto é o suficiente para que, com meios de armazenamento externo suficientes, se consiga fazer qualquer computação, mesmo aquelas realizadas por supercomputadores. Entretanto, a forma de se programar, que na verdade se reduz a acender e apagar diretamente sinais de controle, torna difícil a construção de programas para a solução de problemas mais complexos de transformação de informação.

AS CPUs reais têm características que as distinguem das que vimos aqui, que foram projetadas com propósitos pedagógicos:

- programas, em sua forma de carga na memória do processador, são formados por *instruções*;
- o efeito de cada instrução é obtido pela execução de várias micro-instruções, e correspondem aproximadamente aos textos nas colunas “Efeito Acumulado” dos programas que fizemos para as CPUs anteriores;
- valores literais são codificados nas instruções;
- não existe uma memória para programas e outra para dados; uma única memória RAM abriga dados e programa;
- existem circuitos para sincronização de operações de entrada e saída;
- o processador executa um ciclo de leitura e execução de instruções;
- a programação pode ser feita em *linguagem de montagem*, o que representa um grande avanço com relação à programação por sinais de controle.

Instrução	Efeito
LOAD	Carrega o operando no acumulador
STORE	Armazena o valor do acumulador no endereço de memória especificado pelo operando
ADD	Soma o valor do operando ao valor do acumulador, deixando o resultado no acumulador
COMPARE	Compara o valor do operando com o valor do acumulador
JMP	Desvio incondicional para o endereço especificado pelo operando
JMPEQ	Desvio para o endereço especificado pelo operando se o resultado da última comparação for "iguais"
JMPLT	Desvio para o endereço especificado pelo operando se o resultado da última comparação for "menor que"

Figura 160: Instruções Típicas de uma CPU simples

A Figura 160 mostra alguns exemplos de instruções típicas de CPUs simples. Como você pode reparar comparando com a Figura 159, cada instrução corresponde aproximadamente ao efeito acumulado de uma sequência de micro-instruções.

O processo de programação em assembler consiste em preencher uma tabela usando não os códigos das instruções, mas seus mnemônicos, e também usando nomes (*labels*) dados a posições de memória, e não os endereços efetivos. Isso torna o programa muito mais fácil de se escrever e se ler. Os bits de cada instrução são depois obtidos por uma substituição cuidadosa dos mnemônicos e dos nomes de posições de memória por seus códigos em bits, em um processo que chamamos de *montagem* da instrução.

Label	Size	Address10	Address16	Instruction	Mode	Operand	Comentários
	1	0	00	LOAD	0	0	Zera o acumulador
	1	1	01	STORE	0	SUM	Coloca 0 em SUM
	1	2	02	LOAD	0	X	Carrega o endereço X no acumulador
	1	3	03	STORE	0	P	Coloca o endereço X em P
LOOP	1	4	04	LOAD	1	SUM	Carrega o conteúdo de SUM no acumulador
	1	5	05	ADD	2	P	Soma o conteúdo da posição de memória cujo endereço é P ao acumulador
	1	6	06	STORE	0	SUM	Coloca o resultado na posição SUM
	1	7	07	LOAD	1	P	Carrega o conteúdo de P
	1	8	08	ADD	0	1	Soma 1
	1	9	09	STORE	0	P	Coloca o resultado em P
	1	10	0A	COMPARE	0	XEND	Compara XEND com o acumulador
	1	11	0B	JMPLT	0	FINISH	Se for menor, desvia para FINISH
	1	12	0C	JMP	0	LOOP	Senão, volta para LOOP
FINISH	1	13	0D	OUTPUT	1	SUM	Coloca o resultado na saída
	1	14	0E	HALT			Para.
X	1	15	0F			3142	Números a serem somados
	1	16	10			4542	
	1	17	11			3325	
	1	18	12			1234	
XEND	1	19	13			8786	
SUM	1	20	14			0	
P	1	21	15			0	

Figura 161: Código fonte de um programa para uma CPU simples

Na Figura 161 nós vemos um exemplo de um programa escrito em assembler de uma CPU muito simples quando comparada com CPUs reais, mas muito complicada para os nossos propósitos, a CPU Pipoca, descrita em detalhes no Apêndice A: A CPU Pipoca. Este programa soma os conteúdos das posições de memória com endereços entre 15 e 19 (usando os nomes – os *labels* – endereços entre X e XEND), e coloca o resultado na saída do processador. Vamos explicar somente alguns aspectos deste programa; não se preocupe em entende-lo em detalhe.

Label	Size	Address10	Instruction	Mode	Operand	Comentários
	1	0	LOAD	0	0	Zera o acumulador
	1	1	STORE	0	SUM	Coloca 0 em SUM
	1	2	LOAD	0	X	Carrega o endereço X no acumulador
	1	3	STORE	0	P	Coloca o endereço X em P
LOOP	1	4	LOAD	1	SUM	Carrega o conteúdo de SUM no acumulador
	1	5	ADD	2	P	Soma o conteúdo da posição de memória cujo endereço é P ao acumulador
	1	6	STORE	0	SUM	Coloca o resultado na posição SUM
	1	7	LOAD	1	P	Carrega o conteúdo de P
	1	8	ADD	0	1	Soma 1
	1	9	STORE	0	P	Coloca o resultado em P
	1	10	COMPARE	0	XEND	Compara XEND com o acumulador
	1	11	JMPLT	0	FINISH	Se for menor, desvia para FINISH
	1	12	JMP	0	LOOP	Senão, volta para LOOP
FINISH	1	13	OUTPUT	1	SUM	Coloca o resultado na saída
	1	14	HALT			Para
X	1	15			3142	
	1	16			4542	
	1	17			2325	Números a serem somados
	1	18			234	
XEND	1	19			876	
SUM	1	20			0	
P	1	21			0	

Instruções

Dados

Figura 162: Um programa descreve a ocupação da única memória por instruções e por dados

A primeira coisa a ser percebida é que o programa descreve tanto instruções como dados, como mostra a Figura 162. A cada linha do programa corresponde uma palavra na memória.

No campo de instrução, o programador coloca o mnemônico, e não o código binário de cada instrução. A coluna Label é usada pelo programador para dar nomes a endereços de memória; estes labels podem ser usados na coluna Operando do programa fonte, tornando muito mais fácil modificar um programa quando se quer alterar a ocupação da memória. Na Figura 163 e na Figura 164 você pode ver em destaque dois exemplos do uso de labels no programa fonte.

Label	Size	Address10	Address16	Instruction	Mode	Operand	Comentários
	1	0	00	LOAD	0	0	Zera o acumulador
	1	1	01	STORE	0	SUM	Coloca 0 em SUM
	1	2	02	LOAD	0	X	Carrega o endereço X no acumulador
	1	3	03	STORE	0	P	Coloca o endereço X em P
X	1	15	0F			3142	Números a serem somados
	1	16	10			4542	
	1	17	11			3325	
	1	18	12			1234	
XEND	1	19	13			8786	
SUM	1	20	14			0	
P	1	21	15			0	

Figura 163: Um exemplo de uso de labels como operandos no código fonte

Label	Size	Address10	Address16	Instruction	Mode	Operand	Comentários
LOOP	1	4	04	LOAD	1	SUM	Carrega o conteúdo de SUM no acumulador
	1	5	05	ADD	2	P	Soma o conteúdo da posição de memória cujo endereço é P ao acumulador
	1	6	06	STORE	0	SUM	Coloca o resultado na posição SUM
	1	7	07	LOAD	1	P	Carrega o conteúdo de P
	1	8	08	ADD	0	1	Soma 1
	1	9	09	STORE	0	P	Coloca o resultado em P
	1	10	0A	COMPARE	0	XEND	Compara XEND com o acumulador
	1	11	0B	JMPLT	0	FINISH	Se for menor, desvia para FINISH
	1	12	0C	JMP	0	LOOP	Senão, volta para LOOP
FINISH	1	13	0D	OUTPUT	1	SUM	Coloca o resultado na saída

Figura 164: Outro exemplo de uso de labels como operandos

Cada instrução no programa fonte deve ser codificada em bits para se ter uma imagem da memória gravada em alguma mídia, que será carregada na memória do processador para a execução do programa. Na Figura 165 você pode ver um exemplo deste procedimento de codificação em bits, que é chamado de *montagem* da instrução.

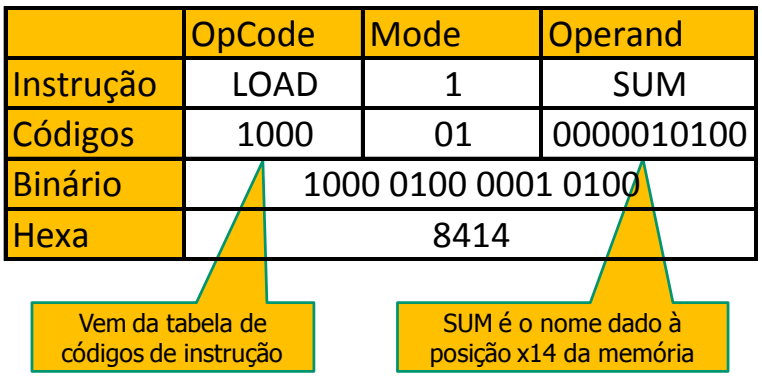


Figura 165: Montagem de uma instrução

A montagem de um programa, ou seja, a tradução do programa fonte para binário, é uma tarefa insana, com fortes exigências de verificação. Mas essa tarefa só foi feita manualmente pelos pioneiros da computação. Cedo se percebeu que os computadores poderiam ser usados não somente para os cálculos de bombas atômicas, que constituíam sua finalidade primária, mas para automatizar o processo de montagem.

Montadores, ou assemblers, são programas que lêm programas, escritos por um programador, e geram arquivos com imagens binárias a serem carregadas na memória. Eles se encarregam da tarefa de substituir mnemônicos e labels pelos bits correspondentes, eliminando erros, e consequentemente a necessidade de verificação da montagem. Para programar a CPU Pipoca nós desenvolvemos uma planilha que se encarrega do processo de montagem de arquivos com imagens da memória.

O *Apêndice A: A CPU Pipoca* descreve com mais detalhes o circuito completo deste processador, assim como uma planilha que pode ser usada para a sua programação, ou para modificar seu micro-programa, modificando ou acrescentando novas instruções.

5 Programas Scilab

It is felt that FORTRAN offers as convenient a language for stating problems for machine solution as is now known.
J.W. Backus, H. Herrick e I.Ziller[40]

5.1 Compiladores, Interpretadores e Sistemas Operacionais

Nós vimos como programar um processador usando micro-instruções, o que é uma tarefa árdua mesmo para transformações de informação extremamente simples como somar dois números. Vimos também, sem entrar em maiores detalhes, que podemos programar um computador usando instruções, o que efetivamente representa um progresso com relação à programação direta por sinais de controle codificados em micro-instruções. Mas isto não é nem de longe uma tarefa confortável para quem tem um problema de tranformação de informação mais ambicioso. Ao construir um programa usando instruções, o programador deve pensar em composições de instruções que refletem a arquitetura da máquina específica em que está trabalhando: seus registradores, memórias, rotas de dados. A tarefa de programar se torna extremamente detalhada e propensa a erros. Além disso, um programa feito para um computador não pode ser executado por um outro com um repertório distinto de instruções.

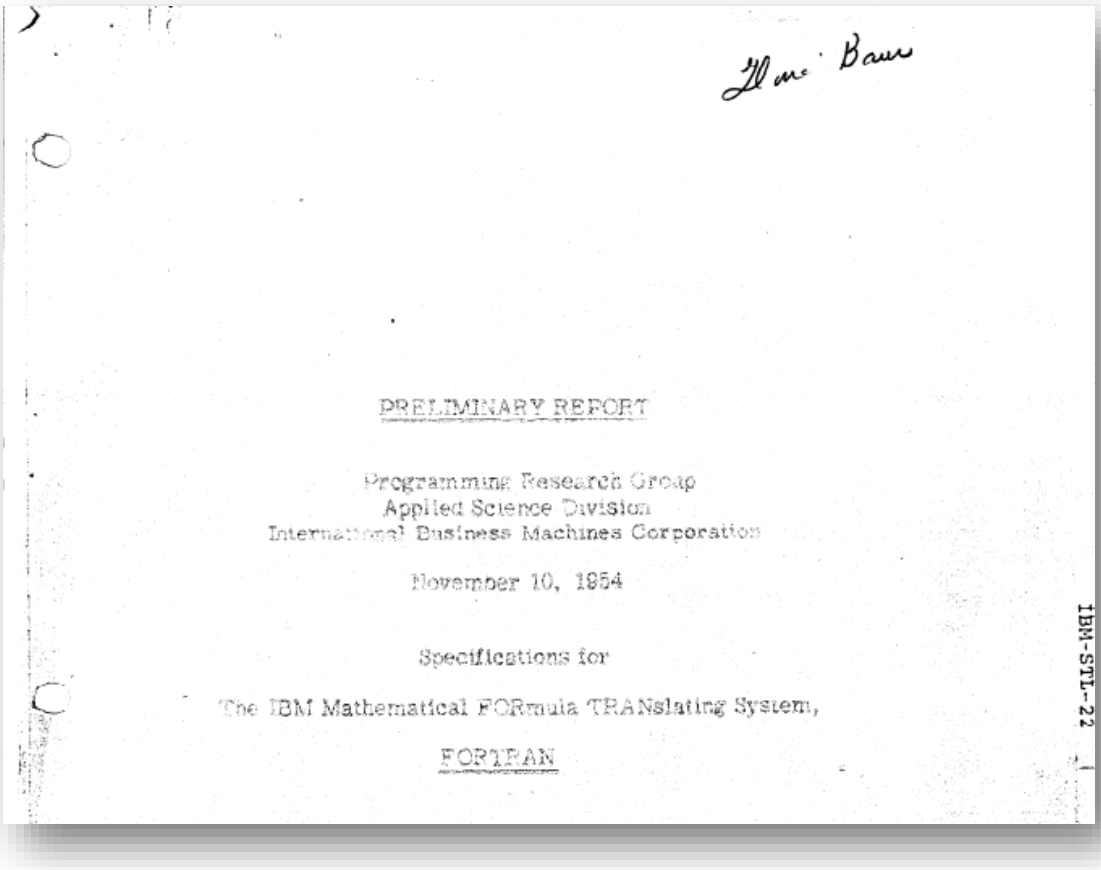
O processo de montagem manual dos bits das instruções de um programa escrito com mnemônicos pode ser feito com a ajuda de um *assembler*, um programa que lê outro programa escrito com mnemônicos e labels, e faz automaticamente a montagem das instruções. Programas montadores melhoraram muito a vida dos programadores, que antes tinham que refazer todo o processo de montagem ao ter, por exemplo, uma posição de memória modificada. A montagem manual de instruções foi feita pelos primeiros programadores, ainda na década de 40. Montadores foram introduzidos no início dos anos 50.

Produzir programas que facilitam a programação é na verdade uma idéia central na ciência e na indústria da computação. Em 1954 a linguagem Fortran – de *Formula Translating* – foi proposta por um grupo de pesquisadores da IBM. É com alguma emoção que nós, cientistas da computação, vemos o fac-símile do original – datilografado, naturalmente – do relatório técnico *Preliminary Report: Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN*[41], mostrado na Figura 166. Fortran é sem qualquer dúvida a mãe de todas as linguagens. Tendo passado por diversas atualizações, Fortran é ainda uma linguagem muito utilizada por cientistas e engenheiros, e isso não deve mudar em um futuro breve.

Um *compilador* é um programa que tem como entrada um outro programa, escrito em uma linguagem de alto nível, e que normalmente produz um programa equivalente ao que recebeu como entrada, escrito em linguagem de máquina para uma arquitetura específica. O primeiro compilador Fortran foi escrito em assembler do IBM 704, uma máquina que tipicamente contava com 15K de memória. Desta época até hoje já foram desenvolvidas com maior ou

[40] John W. Backus et al, “Preliminary Report: Specifications for the IBM Mathematical FORmula TRANSlating System, FORTRAN | Computer History Museum”, 1954, http://www.computerhistory.org/collections/accession/102679231.
[41] Ibidem.

menor grau de sucesso muitas linguagens – milhares delas, literalmente. Um mapa cronológico com as principais linguagens de programação pode ser encontrado na referência[42].



PRELIMINARY REPORT

Programming Research Group
Applied Science Division
International Business Machines Corporation

November 10, 1954

Specifications for

The IBM Mathematical FORmula TRANslating System,

FORTRAN

IBM-STL-22

Figura 166: Fac-símile do relatório de especificação da linguagem FORTRAN[43]

Outras linguagens importantes que são sucessoras diretas de Fortran são:

- Cobol, usada em aplicações comerciais desde 1959,
- a linguagem C, de 1971, que pode produzir programas extremamente eficientes,
- C++, sucedânea de C e que é orientada a objetos, isto é, permite a definição de dados e de operações sobre estes dados de forma muito elegante,
- Basic, criada em 1964 e que bastante tempo depois recebeu grandes investimentos da Microsoft,
- Pascal, de 1970, muito usada como primeira linguagem em cursos de programação,
- Python, de 1991 que é usada na plataforma de programação Google,
- Java, de 1995, que certamente é a linguagem que hoje em dia recebe maiores investimentos da indústria de software, e
- PHP, de 1995, que tem muitos adeptos na comunidade de software livre (o Moodle é escrito em PHP).

[42] "The History of Programming Languages - O'Reilly Media", acessado 7 de março de 2011, http://oreilly.com/news/languageposter_0504.html.
[43] Ibidem.

Existem ainda linguagens que seguem outros paradigmas de programação, como linguagens funcionais, das quais LISP é provavelmente a mais importante, e linguagens lógicas como Prolog.

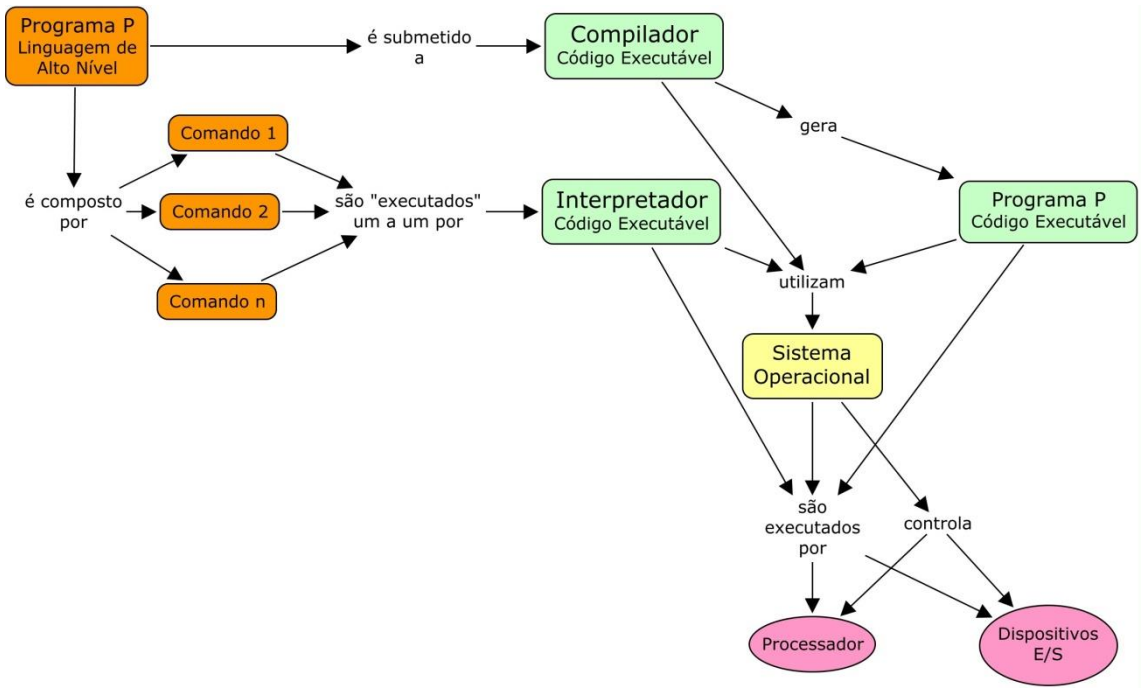


Figura 167: Compiladores, interpretadores e sistema operacional

A execução de programa feito em uma linguagem de alto nível depende de um processo de compilação ou interpretação. A Figura 167 mostra um arranjo típico de componentes de software em um sistema computacional moderno:

- O programador produz um arquivo texto com o programa escrito em uma linguagem de alto nível; esta forma do programa é conhecida como *programa fonte*;
- Dependendo do seu ambiente de desenvolvimento e de execução, o programa fonte pode ser *compilado*, isto é, utilizado como entrada para um compilador, que produz um *código executável* (também chamado de *programa objeto*), armazenado como um outro arquivo, ou pode ser *interpretado*, isto é, utilizado como entrada para um interpretador que se encarrega de produzir diretamente o efeito do programa.
- Programas em linguagem de alto nível são compostos por comandos. Compiladores examinam o programa fonte completo antes de produzir o programa objeto, enquanto interpretadores podem examinar e executar comando por comando do programa fonte.
- Compiladores e interpretadores são programas como quaisquer outros, sendo resultado da compilação de programas fonte.
- Um programa objeto é composto por instruções executáveis pelo processador alvo; para cada arquitetura de processador é necessário adaptar o compilador ou construir um novo. Com isso um mesmo programa fonte pode ser utilizado em computadores com diferentes arquiteturas, desde que existam compiladores de sua linguagem para estas arquiteturas.
- Programas efetivamente especiais são os sistemas operacionais, como Windows, Linux ou MacOS. Sistemas operacionais coordenam o uso do processador, cujo tempo é dividido entre a execução de códigos objeto de diversos programas e também do próprio sistema operacional.
- Programas objeto normais não têm instruções para acesso direto a equipamentos de entrada e saída, como discos ou o monitor. Operações de entrada e saída são

realizadas exclusivamente pelo sistema operacional, que oferece estas operações como serviços utilizados pelos programas comuns. Com isso, o sistema operacional consegue administrar conflitos no uso dos dispositivos de E/S, e também permitir que programas normais possam desfrutar de abstrações confortáveis como arquivos e diretórios, ao invés de terem que lidar com detalhes de cada unidade de disco.

- Isso faz com que compiladores tenham também que ter adaptações para cada sistema operacional. Por exemplo, um compilador C deve ter uma versão para Windows e outra para Linux.
- Uma mesma linguagem pode ser implantada por compilação ou por interpretação. Mesmo se tradicionalmente C e Fortran são linguagens implantadas por compilação, nada impede que interpretadores sejam construídos para estas linguagens, e que compiladores sejam construídos para linguagens tradicionalmente interpretadas como Scilab e Matlab, das quais falaremos a seguir.

5.2 Scilab

Nos fins dos anos 70 Cleve Moler criou uma linguagem, Matlab[44], voltada para o tratamento de matrizes, que, em 1984, foi lançada comercialmente pela empresa MathWorks. Matlab vem de *Matrix Laboratory*, e é um fenômeno de sucesso entre engenheiros e cientistas. O Matlab é um *interpretador*.

Scilab, a linguagem que adotamos neste curso, é desenvolvida desde 1990 por pesquisadores do Institut Nacional de Recherche en Informatique et Automatique, o INRIA, e da École Nationale des Ponts et Chaussées, duas instituições francesas. É muito semelhante ao Matlab e, fator importante para sua escolha, é gratuito. O Scilab é também um ambiente de desenvolvimento e interpretação de programas; a versão utilizada neste texto é a 5.4.0.

Do ponto de vista da ciência da computação, Matlab e Scilab não mereceriam destaque em uma galeria de linguagens de programação. Entretanto, a facilidade que oferecem para a construção de pequenos programas voltados para engenharia e ciência não encontra rival nas linguagens tradicionais como Fortran, C ou Java.

Antes de entrarmos na apresentação do ambiente e da linguagem Scilab queremos colocar algumas observações gerais sobre linguagens de programação, que você deve ter em mente ao iniciar seu estudo. Linguagens de programação são linguagens formais, assim chamadas por contraste com linguagens naturais como português ou inglês. Uma linguagem de programação, como as linguagens naturais, une riqueza de expressão a detalhes sintáticos e algumas arbitrariedades. Detalhes e arbitrariedades frequentemente vêm de escolhas feitas no passado, incluindo algumas que já não fazem mais sentido mas que são mantidas por uma inércia natural. Seu aprendizado exige uma postura paciente, pois envolve no início uma taxa relativamente alta de memorização. Como acontece com as linguagens naturais, a fluência vem com o uso, e com a fluência vem a percepção da riqueza da linguagem.

Como dissemos, Scilab é também um ambiente que interpreta comandos e programas. Ele oferece uma *console* para interação com o usuário, um editor para a construção de programas, o SciNotes, e também emite mensagens de erros relativos tanto à obediência de comandos e programas às regras da linguagem como a problemas que podem ocorrer na execução, como uma divisão por zero.

[44] Mathworks, “MathWorks - MATLAB and Simulink for Technical Computing”.

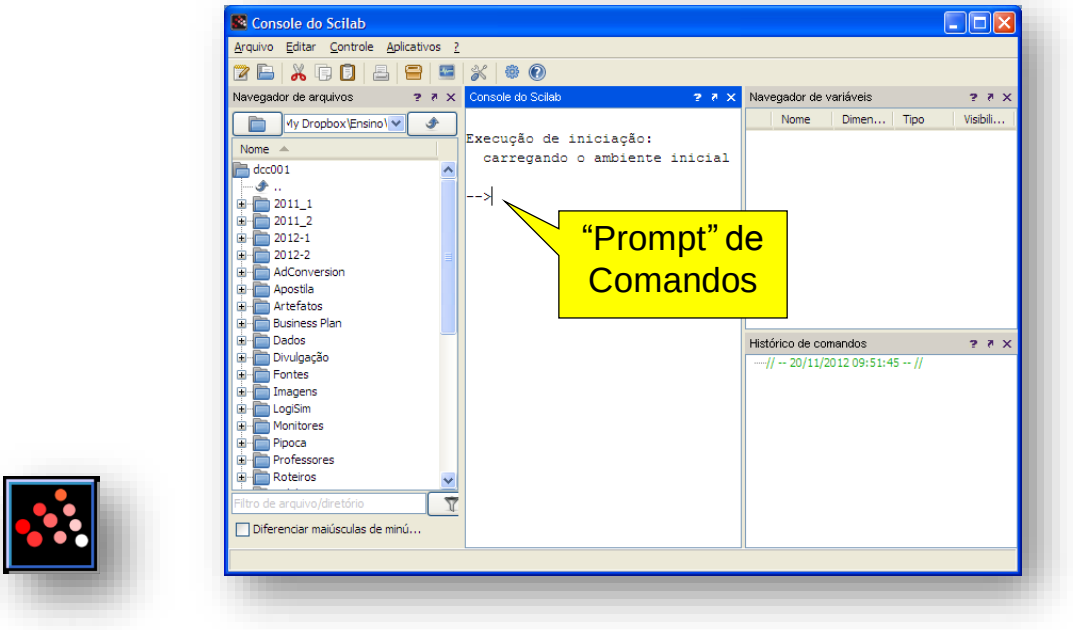


Figura 168: Ícone e tela inicial com a console do Scilab

A Figura 168 mostra a tela obtida ao clicar sobre o ícone do Scilab. É uma tela com quatro painéis, uma barra de menus e uma barra de ferramentas. A janela central, , sobre a qual iremos inicialmente concentrar nossa atenção, é a *console*, com um "prompt " de comandos, indicado pela setinha `-->` . É nesse prompt que são digitados *comandos* a serem interpretados pelo Scilab.

5.3 Variáveis e Comandos de Atribuição

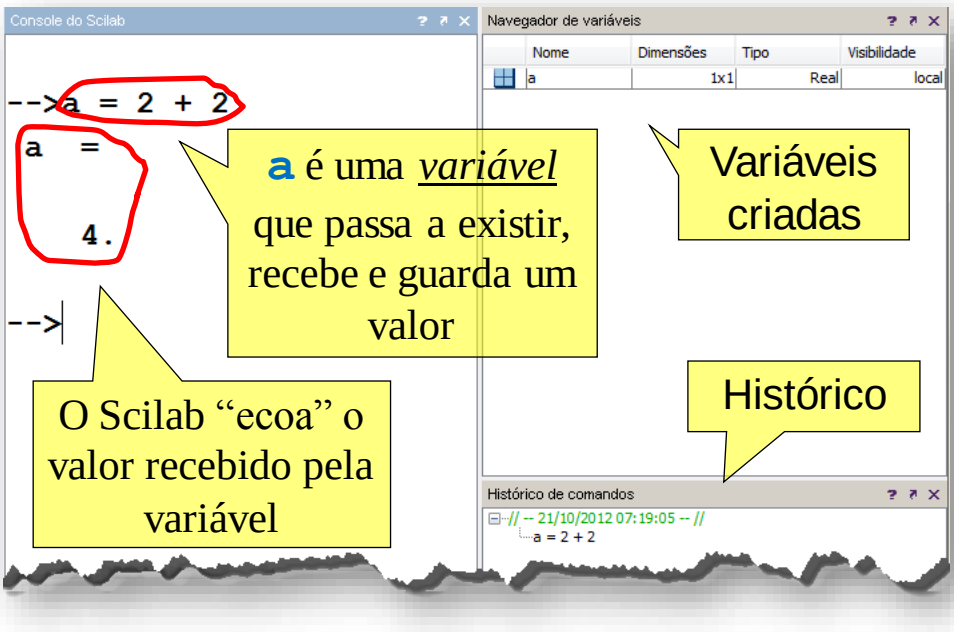

Console do Scilab
Navegador de variáveis
Nome Dimensões Tipo Visibilidade
a 1x1 Real local
-->a = 2 + 2
a =
4.
-->
a é uma variável que passa a existir, recebe e guarda um valor
Variáveis criadas
Histórico
O Scilab "ecoa" o valor recebido pela variável
Histórico de comandos
// -- 21/10/2012 07:19:05 -- //
a = 2 + 2

Figura 169: Um comando de atribuição

O principal comando que transforma informação é chamado *comando de atribuição.* Um primeiro exemplo está na Figura 169, que mostra o efeito no Scilab da digitação na console do comando `a = 2 + 2`, seguida de <enter>. Aqui, `a` é uma *variável* que passa a existir no

interpretador, e que recebe e armazena um valor - no caso, 4. A existência dessa variável é confirmada pelo painel de navegação de variáveis, no canto superior direito da tela. Após executar um comando de atribuição o Scilab "ecoa" o seu valor, isto é, imprime na console o valor recebido pela variável.

Variáveis são nomes para espaços de memória gerenciados pelo Scilab; um programador não precisa ter qualquer idéia de como esse gerenciamento é feito. Variáveis têm seus nomes escolhidos pelo programador segundo algumas regras:

- O primeiro caractere do nome deve ser uma letra, ou qualquer caractere dentre '%', '_', '#', '!', '$' e '?';
- Os outros podem ser letras ou dígitos, ou qualquer caractere dentre '_', '#', '!', '$' e '?'.
- Mesmo sendo um produto francês, o Scilab não permite o uso de letras acentuadas ou de cedilhas em nomes de variáveis.

Exemplos de nomes de variáveis válidos são `a`, `A`, `jose`, `total_de_alunos`, `#funcionarios`. O Scilab distingue entre maiúsculas e minúsculas e, portanto, `a` e `A` seriam variáveis diferentes. Nomes não válidos são `1Aluno` (porque o primeiro caractere é um algarismo), `total de alunos` (porque tem espaços), ou `josé` (porque é acentuado).

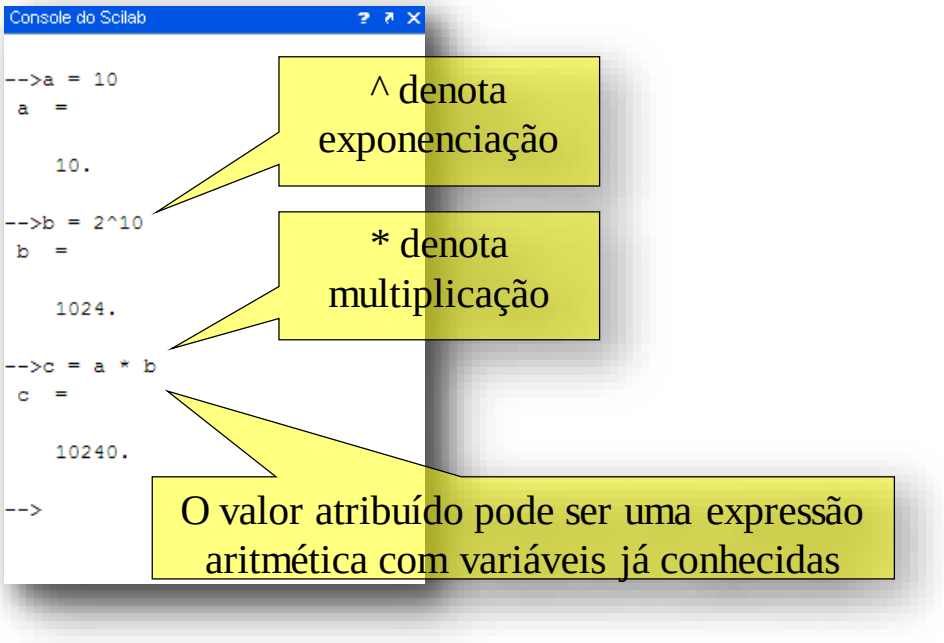


Figura 170: Variáveis e comandos de atribuição

Após a execução dos comandos na Figura 170, o painel de navegação de variáveis teria o formato mostrado na Figura 171, com as variáveis `a`, `b` e `c` criadas.

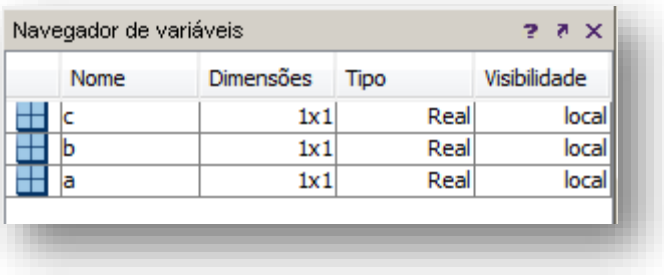


Figura 171: Navegador de variáveis após a execução dos comandos na Figura 170

Um comando de atribuição tem o formato

<variável alvo> = <expressão>

onde:

- A <variável alvo>, se não existia, passa a existir;
- Se existia, o valor anterior é perdido;
- Na execução do comando, a <expressão> é calculada, e o resultado é atribuído à <variável alvo>.

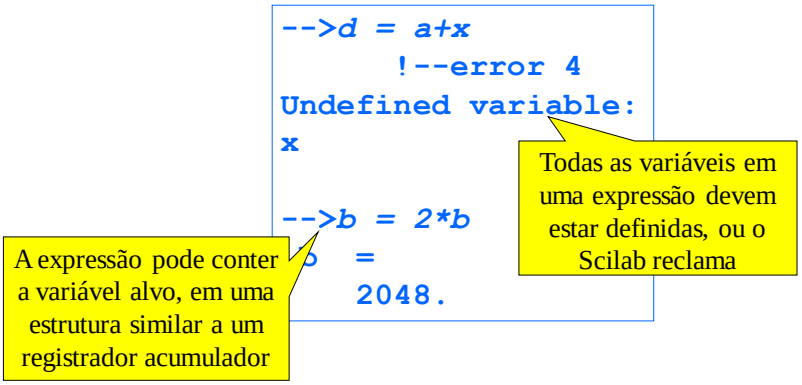

-->d = a+x
!--error 4
Undefined variable:
x
-->b = 2*b
2048.
Todas as variáveis em uma expressão devem estar definidas, ou o Scilab reclama
A expressão pode conter a variável alvo, em uma estrutura similar a um registrador acumulador

Figura 172: Usos e erros em comandos de atribuição

Conforme ilustrado na Figura 172, todas as variáveis envolvidas na <expressão> devem ter valores definidos no momento da execução do comando.

Vale ainda observar que a <expressão> pode conter a <variável alvo>, em um arranjo similar ao utilizado nos registradores acumuladores.

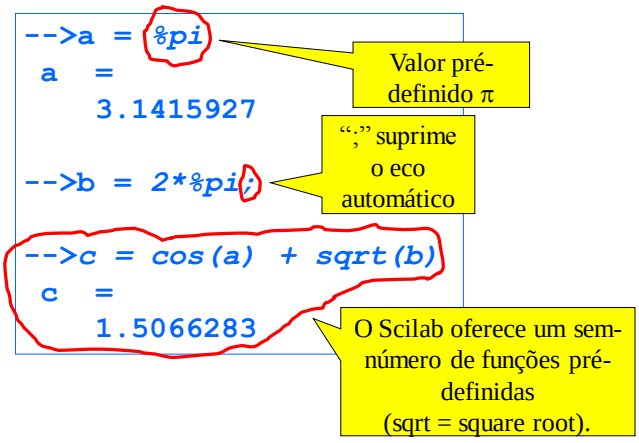


Figura 173: Exemplos de comandos de atribuição com variáveis com valor pré-definido, supressão de eco e funções elementares

O Scilab oferece também variáveis com valores pré-definidos, como %pi ou %e, e uma enorme variedade de funções pré-definidas (veja a Figura 174). Valores numéricos são representados no Scilab em ponto flutuante de 64 bits; %pi é forma que temos para, em um programa, nos referirmos à melhor aproximação de π nessa representação; o mesmo vale para %e e outras constantes.

Um detalhe muito útil é a possibilidade de supressão do eco automático, que algumas vezes polui a tela com informação desnecessária, e que é obtido com o uso de um "**;**" colocado após o comando de atribuição.

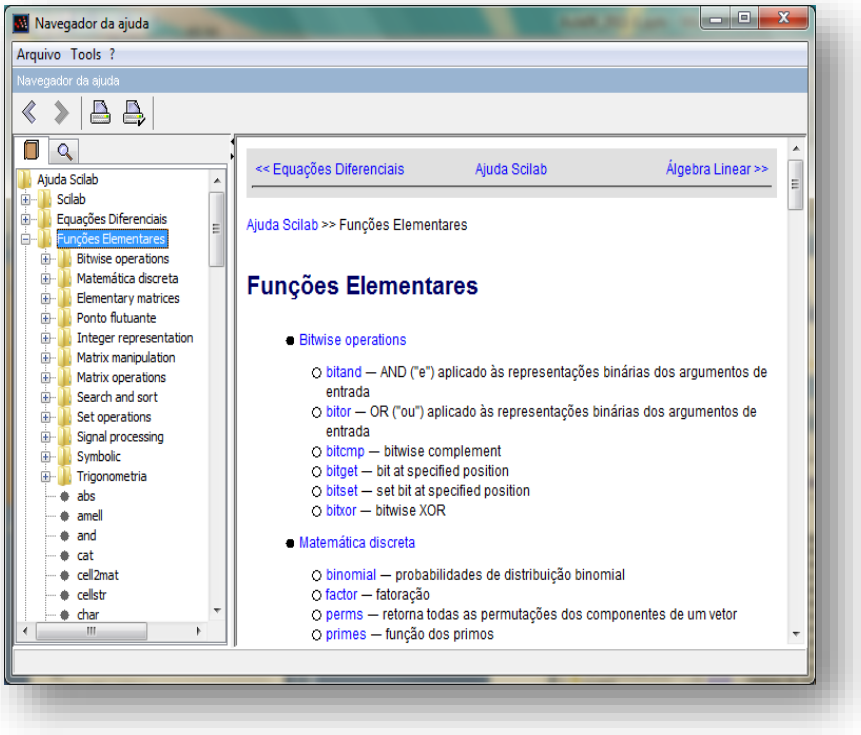


Figura 174: Lista de funções elementares encontrada no help do Scilab

A construção e a compreensão de expressões aritméticas mais elaboradas exige o conhecimento das regras de prioridades entre operadores e o uso de parênteses para se obter o resultado desejado. Como um exemplo, qual valor será atribuído a uma variável `x` pelo comando `x = 2^3*4`, o valor $2^{3\times4} = 2^{12} = 4096$, ou o valor $2^3 \times 4 = 8 \times 4 = 32$? A Figura 175 mostra as prioridades empregadas pelo Scilab no cálculo de expressões. Ali vemos que a potenciação tem prioridade sobre a multiplicação, e portanto o valor atribuído a `x` pelo comando acima será $2^3 \times 4 = 8 \times 4 = 32$.

Prioridade	Operação
1ª	Potenciação
2ª	Multiplicação, divisão
3ª	Adição, subtração

Figura 175: Prioridades de operadores aritméticos

Se a intenção do programador era atribuir a `x` o valor $2^{3\times4} = 2^{12} = 2048$, parênteses deveriam ter sido usados no comando de atribuição, como em `x = 2^(3*4)` .

```
-->2^3*4
 ans  =
32.
--
>2^(3*4)
 ans  =
4096.
-->2^3^4
 ans  =
2.418D+24
--
>2^(3^4)
 ans  =
2.418D+24
--
>(2^3)^4
 ans  =
4096.
-->2*3+4
 ans  =
10.
--
>2*(3+4)
 ans  =
14.
```

Figura 176: Prioridades e parênteses influenciando o valor de uma expressão aritmética

Na Figura 176 nós vemos alguns exemplos de como o Scilab interpreta expressões aritméticas. De uma forma geral, é recomendável o uso de parênteses para tornar clara a ordem em que o programador deseja que as operações de uma expressão sejam realizadas.

Em Scilab (como em Fortran, C, Java, e praticamente todas as linguagens de programação) não se usam chaves ou colchetes; somente parênteses são usados, e a cada parênteses aberto deve corresponder um parênteses fechado. Toda expressão deve ser escrita de forma linear; por exemplo,

$$\frac{-b+\sqrt{b^2-4ac}}{2a}$$

em Scilab assume a forma `(-b+sqrt(b^2 - 4*a*c))/(2*a)`, onde usamos cores para ressaltar a correspondência entre parênteses abertos e fechados.

5.4 Usando o Scilab como Calculadora

Para mostrar a utilidade de um programa, vamos primeiramente resolver com a console Scilab uma equação de segundo grau, com a forma que conhecemos desde a escola secundária:

$$ax^2 + bx + c = 0$$

Nós queremos calcular as raízes para $a = 534.2765$, $b = 9987.3431$ e $c = 225.7690$. Nós sabemos que as raízes são encontradas pelas fórmulas

$$r_1 = \frac{-b+\sqrt{\Delta}}{2a}$$

e

$$r_2 = \frac{-b-\sqrt{\Delta}}{2a}$$

onde $\Delta = b^2 - 4ac$.

Temos aqui um problema de transformação de informação: dos valores de a, b e c, queremos obter valores para r_1 e r_2. Em todo problema de transformação de informação, mesmo antes de tentar resolvê-lo, temos que pensar em testes que verifiquem a correção do que consideramos ser uma solução. No caso, dispomos de um teste simples: valores r_1 e r_2 que venham a ser encontrados devem ser tais que o valor calculado de $f(x) = ax^2 + bx + c$ seja igual a zero nesses pontos.

A primeira coisa que faremos é inicializar variáveis `a`, `b` e `c`, conforme mostrado na Figura 177.

```
-->a =
534.2765
 a   =

534.2765
-->b =
9987.3431
 b   =

9987.3431
-->c =
225.7690
 c   =
    225.769
```

Figura 177: Inicialização dos coeficientes de uma equação de 2o grau

Depois, calculamos Δ e as raízes, como mostrado na Figura 178.

```
-->delta = b^2 - 4*a*c
 delta  =
    99264530.
-->r1 = (-b+sqrt(delta))/(2*a)
 r1  =
  - 0.0226329
-->r2 = (-b-sqrt(delta))/(2*a)
 r2  =
  - 18.670578
```

Figura 178: Cálculo das raízes da equação de 2o grau

Repare que usamos variáveis `delta`, `r1` e `r2`, com nomes aceitáveis para o Scilab e, principalmente, com significado para nós. Repare também nas expressões usadas nos comandos de atribuição. Erros comuns cometidos por iniciantes são:

- escrever `delta = b^2 - 4ac`, omitindo os operadores de multiplicação, que entretanto são imprescindíveis para que o Scilab "compreenda" a expressão, ou
- escrever `r1 = (-b+sqrt(delta))/2*a`, o que na verdade levaria ao cálculo de $r_1 = \left(\frac{-b+\sqrt{\Delta}}{2}\right).a$, o que não é o que queremos.

O primeiro erro é menos grave, pois o Scilab irá detetar que existe algo de errado com o comando, e irá manifestar seu estranhamento por uma mensagem como a da Figura 179. Este tipo de erro é conhecido como *erro de sintaxe*.

```
-->delta = b^2 - 4ac
                  !--error 276
Operador, comma, ou semicolon faltante.
```

Figura 179: Mensagem de erro de sintaxe detetado pelo Scilab

O Scilab não consegue entretanto identificar erros do segundo tipo, conhecidos como *erros de semântica*. A sintaxe da expressão está perfeita, mas seu significado – sua semântica – não é aquela que desejávamos. Por isso, sempre é bom verificar os resultados de um cálculo ou de um programa. Para a nossa equação de segundo grau podemos também usar o Scilab para a verificação, com os comandos mostrados na Figura 180. Nesta figura você deve reparar:

- na aparição da variável **`ans`**, que é utilizada pelo Scilab para armazenar resultados de expressões que não fazem parte de um comando de atribuição;
- na notação **`3.865D-12`**, que é a forma de se escrever o valor literal 3.865×10^{-12}.

Ali vemos que o valor do polinômio da equação nas raízes que calculamos não é exatamente zero. Isso não deve constituir preocupação, pois os valores são relativamente muito pequenos, da ordem de 10^{-12} para **`r1`**, e 10^{-13} para **`r2`**. Números no Scilab são armazenados como ponto flutuante de 64 bits (veja a Seção 2.1), onde as operações podem envolver arredondamentos.

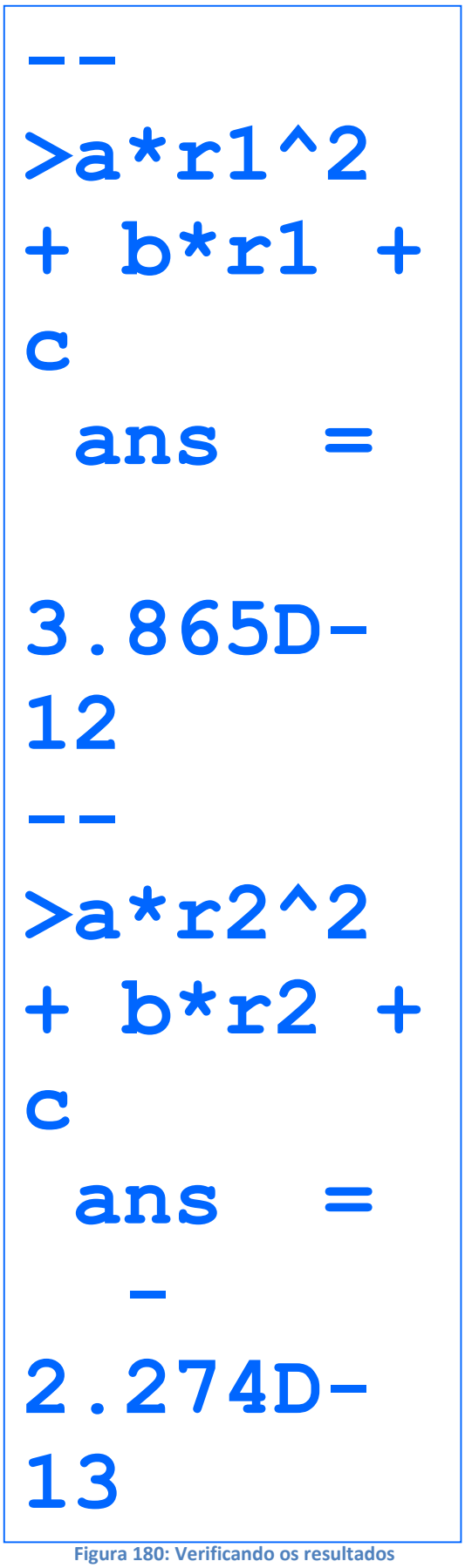


Figura 180: Verificando os resultados

Muito bem, conseguimos usar o Scilab para resolver uma equação de 2º grau, o que não chega a ser uma façanha. Mas tivemos ganhos com relação à execução dos mesmos cálculos com uma calculadora de mão:

- o uso de variáveis evita re-digitações e possíveis erros;
- resultados intermediários são memorizados e podem ser reaproveitados;
- o uso de fórmulas como na Figura 178 aumenta muito a confiança nos cálculos.

As limitações do uso direto da console Scilab para cálculos tornam-se claras quando queremos resolver outra equação de 2º grau. Fórmulas têm que ser re-digitadas, abrindo ocasião para erros, com pouco aproveitamento do trabalho já feito. A solução para isso é usar o Scilab como um interpretador de *programas*.

5.5 Programas Scilab

Um *programa fonte* Scilab é um arquivo texto cujo nome tem a terminação `.sce`. Um arquivo-programa contém comandos Scilab, e é construído usando o editor SciNotes (veja na Figura 181 como o editor é aberto no Scilab 5.1). A execução (interpretação) de um programa se faz seguindo o menu *Executar* da console do SciNotes; essa execução equivale à digitação na console dos comandos presentes no arquivo.

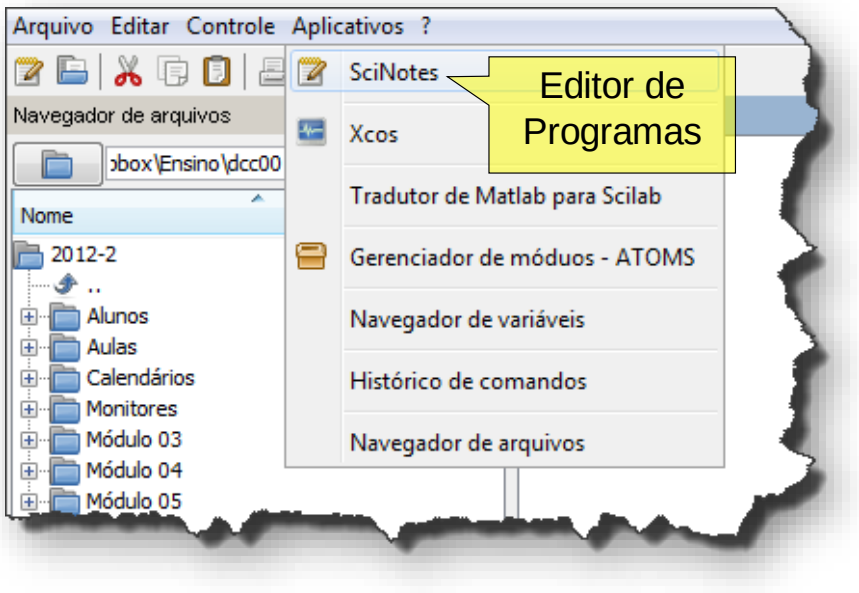


Figura 181: Abrindo o editor SciNotes

Atenção: nunca use o Word ou qualquer outro editor de textos sofisticado para abrir e/ou editar arquivos de programas. Esses editores podem introduzir bits de informação de formatação, o que perturba completamente a interpretação do programa pelo Scilab.

No editor SciNotes você pode:

- Criar um novo programa, através do menu *Arquivo/Novo*;
- Abrir para edição um programa já existente, através do menu *Arquivo/Abrir*
- Editar um programa
- Salvar o programa editado, através do menu *Arquivo/Salvar*
- Executar um programa, através do menu *Executar*

e várias outras coisas que você irá aprendendo aos poucos.

O Scilab trabalha com um *diretório corrente*, que é a primeira opção de localização para procurar e para salvar arquivos. Na console do Scilab você pode escolher o diretório corrente através do menu *Arquivo/Alterar o Diretório Atual*, ou clicando duas vezes sobre um diretório no painel de navegação de arquivos, que sempre mostra o diretório corrente no ponto mais alto da hierarquia exibida. O diretório corrente do SciNotes é o diretório corrente do Scilab no momento em que o editor é aberto.

Um conselho: organize os seus arquivos! Perde-se muito tempo procurando arquivos gravados – aonde mesmo?. O autor destas linhas cria um diretório para cada módulo do curso, onde são colocados todos os arquivos que ali são utilizados; você pode adotar uma organização similar em seu computador pessoal. Ao usar computadores compartilhados, crie um diretório de trabalho com o seu nome, o que irá facilitar a sua "limpeza" posterior.

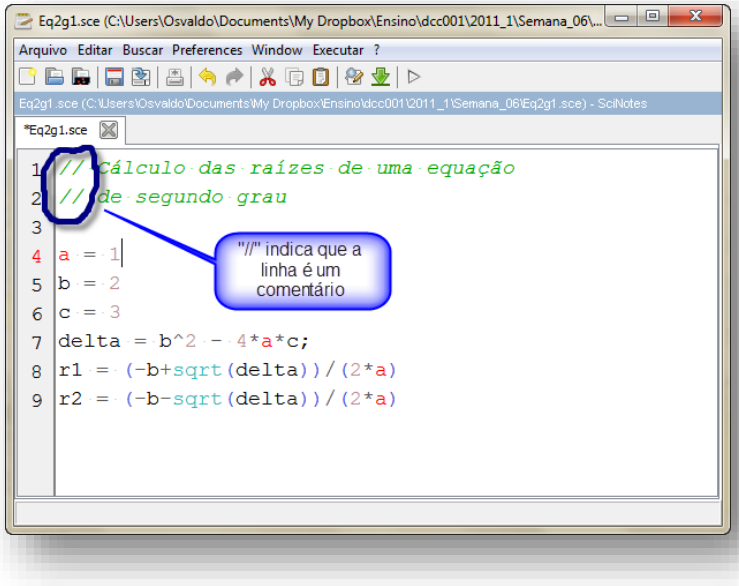

Eq2g1.sce (C:\Users\Osvaldo\Documents\My Dropbox\Ensino\dcc001\2011_1\Semana_06\...
Arquivo Editar Buscar Preferences Window Executar ?
Eq2g1.sce (C:\Users\Osvaldo\Documents\My Dropbox\Ensino\dcc001\2011_1\Semana_06\Eq2g1.sce) - SciNotes
*Eq2g1.sce
// Cálculo das raízes de uma equação
// de segundo grau
a = 1
b = 2
c = 3
delta = b^2 - 4*a*c;
r1 = (-b+sqrt(delta))/(2*a)
r2 = (-b-sqrt(delta))/(2*a)
"//" indica que a linha é um comentário

Figura 182: O SciNotes editando o programa Eq2g1.sce

A Figura 182 mostra um programa que tem em cada linha exatamente os mesmos comandos que utilizamos na console para resolver a equação de 2º grau. Nós demos a este programa o nome Eq2g1.sce; usamos um número no nome do arquivo porque faremos outras versões deste mesmo programa.

As duas primeiras linhas do programa Eq2g1.sce se iniciam por "**//**", o que faz com que sejam ignoradas pelo Scilab no momento da execução. Essas linhas são *comentários*, e têm o importantíssimo objetivo de melhorar a compreensão de um programa por um leitor humano.

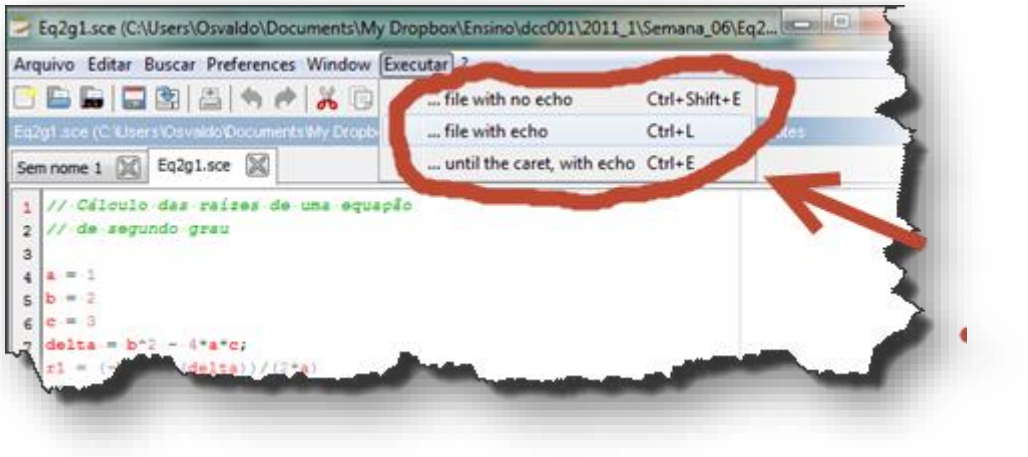

Eq2g1.sce (C:\Users\Osvaldo\Documents\My Dropbox\Ensino\dcc001\2011_1\Semana_06\Eq2...
Arquivo Editar Buscar Preferences Window Executar ?
... file with no echo Ctrl+Shift+E
... file with echo Ctrl+L
... until the caret, with echo Ctrl+E
Sem nome 1
Eq2g1.sce
// Cálculo das raízes de uma equação
// de segundo grau
a = 1
b = 2
c = 3
delta = b^2 - 4*a*c;

Figura 183: Executando um programa a partir do editor SciNotes

Para executar o programa, você pode usar o menu *Executar/...file with no echo* ou *Executar/...file with echo*, conforme mostra a Figura 183. Experimente as duas possibilidades, e veja os resultados na console do Scilab.

Com um programa gravado em um arquivo, se quisermos resolver uma nova equação, bastará substituir no programa os valores dos novos coeficientes e executá-lo novamente. Comparando com o processo de resolução via console, o uso de um programa reduz consideravelmente o trabalho e as chances de erros de digitação.

Entretanto, a prática de se alterar programas para introduzir dados que se modificam a cada execução não é recomendável, e nem exeqüível quando o volume de dados é muito grande. O melhor a fazer é modificar o programa para permitir que o usuário defina os valores dos coeficientes a cada execução.

```
// Cálculo das
raízes de uma
equação
// de 2o grau
a =
input("Digite o
valor de a:")
b =
input("Digite o
valor de b:")
c =
input("Digite o
valor de c:")
delta = b^2 -
4*a*c
r1 = (-
b+sqrt(delta))/
(2*a)
r2 = (-b-
sqrt(delta))/(2
*a)
```

Figura 184: O programa Eq2g2.sce com os comandos de diálogo em destaque

O comando **`input`** permite essa interação com o usuário em tempo de execução. Como vemos na Figura 184, este comando recebe como parâmetro uma frase a ser exibida para o usuário, que normalmente é usada para solicitar o valor a ser digitado.

```
Digite o
valor de
a:1
 a   =
    1.
Digite o
valor de
b:2
 b   =
    2.
Digite o
valor de
c:3
 c   =
    3.
 delta  =
  - 8.
 r1   =
  - 1. +
1.4142136i
 r2   =
  - 1. -
1.4142136i
```

Figura 185: Execução do programa Eq2g2.sce

A Figura 185 mostra a console do Scilab em uma execução do programa Eq2g2.sce, onde você pode verificar o efeito da execução dos comandos `input`. Os valores digitados pelo usuário para os coeficientes `a`, `b` e `c` foram, respectivamente, 1, 2 e 3. Estes valores levam a um Δ

negativo, e o exemplo serve também para ilustrar a naturalidade com que o Scilab trata números complexos.

5.6 Valores Lógicos e Strings

Uma variável Scilab pode armazenar também valores lógicos correspondentes a verdadeiro e falso, denotados pelos literais Scilab %t e %f (true e false), ou %T e %F. A Figura 186 mostra um exemplo de atribuição de valores lógicos a variáveis usando a console do Scilab.

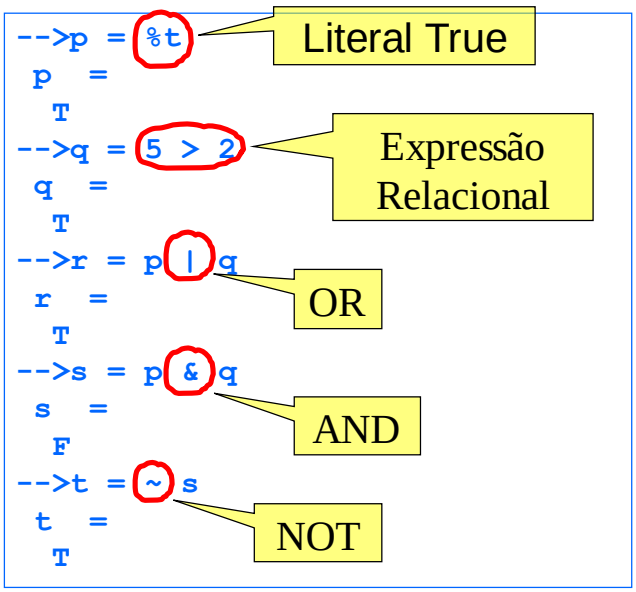


Figura 186: Atribuicão de valores lógicos a variáveis na console do Scilab

Variáveis com valores lógicos podem ser parte de expressões lógicas, que usam os operadores lógicos ~ (NOT), & (AND) e | (OR), definidos, como você pode esperar, exatamente como na Figura 57 (página 48).

>	maior que
>=	maior ou igual a
<	menor que
<=	menor ou igual a
==	igual a
<> ou ~=	diferente de

Figura 187: Operadores relacionais

Expressões lógicas frequentemente fazem uso de comparações entre valores de expressões aritméticas para obter um valor lógico. A Figura 187 mostra os operadores relacionais usados no Scilab, onde você pode reparar que "igual a" é representado por dois "=" adjacentes, uma herança da linguagem C, e que existem duas formas de representação de "diferente de".

```
-->a = 5 > 7
 a   =
  F

-->b = (2+1) == 3
 b   =
  T

-->c = (4-3) <= 0
 c   =
  F
```

Figura 188: Exemplos de expressões lógicas utilizando operadores relacionais

Além de valores lógicos, variáveis Scilab podem armazenar outros dados não numéricos. Na Figura 189 nós vemos exemplos de como atribuir sequências de caracteres – o termo usado é em inglês, *strings* – a variáveis.

```
-->a = "Programação"
 a   =
 Programação
-->b = " de '
 b   =
  de
-->c = "Computadores"
 c   =
 Computadores
```

Aspas simples (') e duplas (") são equivalentes

Figura 189: Atribuindo strings a variáveis

Strings são escritos entre aspas, simples ou duplas. Você pode mesmo iniciar um string com aspas duplas e terminá-lo com aspas simples.

Para strings, + significa concatenação

```
-->a = 'Programação';
-->b = ' de ';
-->c = 'Computadores';

-->Disciplina = a + b + c
 Disciplina  =
 Programação de Computadores
```

Figura 190: Concatenação de strings

Uma operação comum com strings é a *concatenação*, que consiste na justaposição de dois strings. No Scilab a concatenação utiliza o mesmo símbolo da adição numérica, o "+".

Fim do string?

```
-->x = 'String "com aspas"'
                !--error 276
Missing operator, comma, or semicolon
```

Figura 191: Erro ao tentar representar um string contendo aspas

Aspas são usadas para marcar o início e o fim de strings, e isto pode provocar um pequeno problema ao se tentar representar strings que contêm aspas, como mostrado na Figura 191. Para conseguir isso, basta colocar duas aspas consecutivas na posição desejada, como na Figura 192.

```
-->x = 'String ""com aspas duplas""'
 x  =
 String "com aspas duplas"
-->x = 'String ''com aspas simples'''
 x  =
 String 'com aspas simples'
```

Figura 192: Strings contendo aspas

Strings formados por algarismos não são números para o Scilab. O string `'3.1415926'` é na verdade armazenado como uma sequência dos caracteres ASCII ou UTF-8 “3”, “.”, “1”, “4”, etc., e não como um número de ponto flutuante, como mostrado na Figura 55. Se tentarmos realizar operações aritméticas com strings de algarismos, o Scilab irá emitir uma mensagem de erro apropriada:

```
-->2*'3.1415926'"
    !—error 144
Undefiled operation for the given operands
```

Figura 193: Strings formados por algarismos não são números para o Scilab

Se uma variável **s** contém um string em um formato numérico (o que inclui strings como `"-2.3D-23"`), o comando **x = eval(s)** coloca em **x** o valor de **s** convertido para um número codificado em ponto flutuante. A função **isnum(s)** pode ser usada para verificar se s contém um string válido para a conversão

Se **x** contém um valor em ponto flutuante, a função **string(x)** pode ser usada para obter sua conversão para um string com algarismos, sinais e expoentes de um número decimal com valor aproximadamente igual ao valor de **c**. O comando **printf**, que veremos a seguir, permite converter um valor em ponto flutuante com um maior controle sobre seu formato.

Strings também podem ser lidos pelo comando **input**, como no exemplo **Nome = input("Seu nome, por favor")**. Escrevendo dessa forma o comando **input** o usuário deve digitar o seu nome entre aspas. É possível eliminar a necessidade de escrita do string entre aspas usando o comando **input** com um parâmetro extra, um string com o valor **"string"**, como no comando **input("Seu nome, por favor","string")**.

```
-->Nome = input("Seu nome: ")
Seu nome: Jose
Undefined variable: Jose
Seu nome: "Jose"
 Nome  =
 Jose

-->Nome = input("Seu nome: ","string")
Seu nome: Jose
 Nome  =
 Jose
```

Figura 194: Exemplos de uso do comando input na console do Scilab

5.7 Os comandos `if-then-else` e `printf`

Para enriquecer nosso repertório de comandos Scilab, vamos agora construir um terceiro programa que resolve equações do 2º grau, mas com as seguintes alterações na especificação:

- o programa só deverá calcular as raízes quando elas forem reais;
- a saída do programa deverá ser uma frase como "As raízes são xxxx e xxxx", quando as raízes forem reais e, senão, "As raízes são complexas."

```
a = input("Valor de a: ");
b = input("Valor de b: ");
c = input("Valor de c: ");
delta = b^2 - 4*a*c;
if delta >= 0 then
    r1 = (-b+sqrt(delta))/(2*a);
    r2 = (-b-sqrt(delta))/(2*a);
    printf("As raízes são %6.3f e %6.3f",r1,r2);
else
    printf("Raízes complexas");
end
```

Figura 195: O programa Eq2g3.sce

A Figura 195 mostra o programa Eq2g3.sce, que atende a essas especificações. Repare que estamos usando "`;`" após vários dos comandos de atribuição, o que suprime o eco do Scilab e torna a saída mais limpa.

```
if <condição> then
  <bloco "então">
else
  <bloco "senão">
end
```

Figura 196: O comando if

Este programa introduz dois novos comandos: `if` e `printf`. O comando `if` é usado para prescrever comportamentos condicionais na execução do programa. Sua forma geral está mostrada na Figura 196, onde:

- `if`, `then`, `else` e `end` são as *palavras-chave* que o Scilab usa para reconhecer o comando;
- `if` marca o início do comando;
- `<condição>` é uma *expressão lógica*, tipicamente uma comparação entre expressões aritméticas, cujo valor é avaliado como verdadeiro ou falso;
- `then` separa a `<condição>` do `<bloco "então">`;
- `<bloco "então">` e `<bloco "senão">` são conjuntos arbitrários de comandos Scilab;
- `else` marca o fim do `<bloco "então">` e o início do `<bloco "senão">`;
- `end` é a palavra-chave que fecha o comando `if`.
- na execução do comando `if`, o `<bloco "então">` é executado se e somente se a `<condição>` for verdadeira, e o `<bloco "senão">` é executado se e somente se a `<condição>` for falsa.

Em alguns casos não desejamos executar nenhum comando no caso da **<condição>** ser falsa, e o comando pode assumir uma forma simplificada, sem a cláusula else, como mostrado na Figura 197.

```
if <condição> then
  <bloco "então">
else
  // Nenhum comando aqui
End
```

```
if <condição> then
  <bloco "então">
End
```

Figura 197: Duas formas equivalentes do comando if, a da direita sem a cláusula else

A Figura 198 mostra os blocos de comandos e a condição do comando **if** do programa Eq2g3.sce.

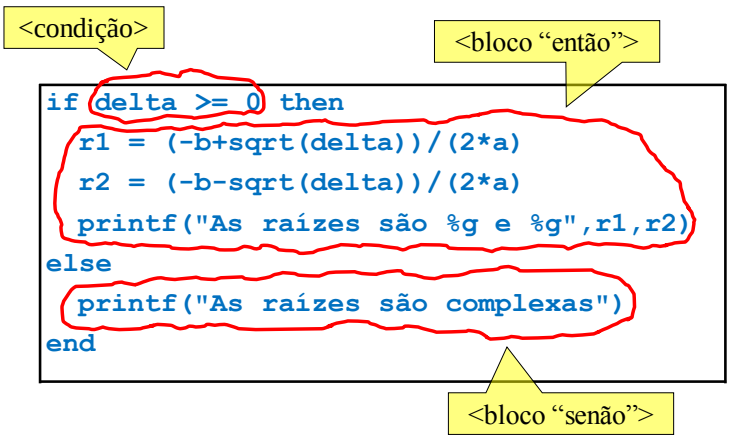


Figura 198 :Partes do comando if do programa Eq2g3.sce

O comando **printf** tem a forma

printf(<frase>,<lista de expressões>)

onde

- **<frase>** é a sentença que se quer imprimir na tela, e é um string que pode estar entremeado por *códigos de formato* como **%g**; **%g** é um código de formato geral para expressões com valores numéricos (nós veremos em seguida expressões com outros tipos de valores); existem vários outros códigos de formato como **%d**, **%f** , **%e** ou **%s**, que iremos explorar em exercícios e em outros exemplos neste texto.
- **<lista de expressões>** é uma lista de expressões separadas por vírgulas, que são calculadas no momento da execução do comando;
- os valores resultantes das expressões na lista são mapeadas um a um nos códigos de formato, na mesma sequência em que aparecem na **<frase>**, e a sentença impressa é obtida pela substituição do valor da expressão na posição marcada pelo código de formato.

No comando **printf("As raízes são %g e %g",r1,r2)** as duas expressões na lista são muito simples, formadas por uma variável. A expressão **r1** é mapeada no primeiro **%g**, e a expressão **r2** é mapeada no segundo **%g**. A Figura 199 mostra uma saída do programa Eq2g3.sce onde se pode ver o efeito da execução deste comando.

```
Digite o valor de a:3
Digite o valor de b:4
Digite o valor de c:1
 delta  =
    4.
 r1  =
  - 0.3333333
 r2  =
  - 1.
As raízes são -0.333333 e -1
```

Figura 199: Uma saída do programa Eq2g3.sce

Vamos agora exercitar nossas novas habilidades fazendo um programa que:

- Leia o nome do aluno, que responde, por exemplo, "José";
- Leia também o total de pontos obtidos pelo aluno;
- Imprima, conforme o caso, a frase *<aluno>, com <pontos> você passou!*, ou então, caso o aluno não tenha obtido um mínimo de 60 pontos, a frase *<aluno>, com <pontos> você não passou!* Exemplos seriam *José, com 80 pontos você passou!*, ou *José, com 40 pontos você não passou!*

```
// Leitura do nome do aluno
Nome = input("Seu nome, por favor:");

// Leitura dos pontos obtidos
Pontos = input(Nome + ", quantos pontos você conseguiu?");

// Decisão e impressão do resultado
if Pontos >= 60 then
   printf("%s, com %g pontos você passou!",Nome,Pontos);
else
   printf("%s, com %g pontos você não passou!",Nome,Pontos);
end
```

Figura 200: O programa PassouNaoPassou.sce

A Figura 200 mostra o programa PassouNaoPassou.sce que atende a esta especificação.

```
Seu nome, por favor:"Maria"
Maria, quantos pontos você conseguiu?90
Maria, com 90 pontos você passou!

Seu nome, por favor:"Jose"
Jose, quantos pontos você conseguiu?47
Jose, com 47 pontos você não passou!
```

Figura 201: Duas execuções do programa PassouNaoPassou.sce

Dois exemplos de execuções deste programa estão na Figura 201. Neste programa é importante observar:

- A frase utilizada no comando `input` para a variável `Pontos` é o resultado de uma operação de concatenação (+);
- Os comandos `printf` utilizam o código de conversão `%s`, apropriado para strings.

Estes dois truques são exemplos de manipulação de strings que podem tornar mais simpática a interação de um programa com seu usuário.

6 Matrizes

Matrizes no Scilab são variáveis que contêm um número potencialmente grande de valores. É na manipulação de matrizes que o Scilab (seguindo o Matlab) mostra uma grande superioridade sobre linguagens como C ou Fortran.

6.1 Matrizes e Comandos de Atribuição

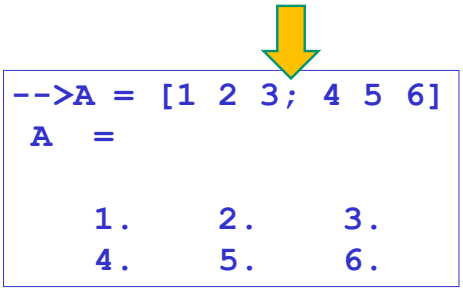


Figura 202: Atribuindo valores literais a uma matriz

A Figura 202 mostra uma maneira simples de se criar uma matriz através de um comando de atribuição na console do Scilab. Os valores dos elementos da matriz são dispostos entre colchetes. Espaços (poderiam ser vírgulas) separam elementos, e "`;`" separa linhas.

É possível extrair o valor de um elemento específico da matriz, designado por seus índices entre parênteses, como mostrado na Figura 203, e também podemos atribuir um valor a um elemento específico de uma matriz, como mostrado na Figura 204.

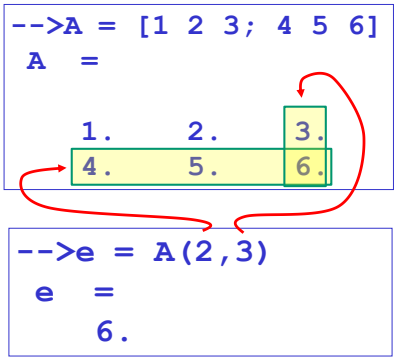


Figura 203: Obtendo o valor de um elemento de uma matriz

`A(2,3)` , por exemplo, refere-se ao elemento na segunda linha e na terceira coluna; `A(1,2)` é o elemento na primeira linha e na segunda coluna.

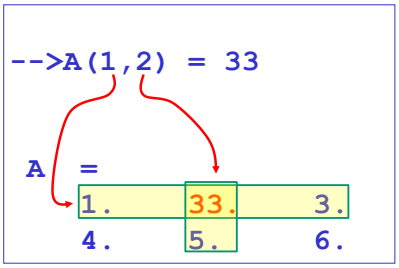


Figura 204: Atribuindo um valor a um elemento de uma matriz

O Scilab é tão orientado para matrizes que *todas* as variáveis Scilab são matrizes. As variáveis simples com que temos trabalhado são, na verdade, matrizes com uma única linha e uma única coluna. É possível perceber isso usando o comando `[nl,nc] = size(A)` para obter as dimensões de uma matriz `A`. A função `size` retorna dois parâmetros, o número de linhas e o número de colunas da matriz. A Figura 205 mostra dois exemplos de uso da função `size`.

```
-->A = [1 2 3; 4 5 6];

-->[nl,nc] = size(A)
 nc  =
    3.
 nl  =
    2.

-->k = 0;

-->[L,C] = size(k)
 C  =
    1.
 L  =
    1.
```

Figura 205: Matrizes e a função size

Uma matriz "cresce" quando atribuímos valores a elementos com índices superiores aos índices já referenciados. Por exemplo, quando fazemos `x = 7` estamos criando uma matriz 1 x 1; se em seguida fizermos `x(2,3) = 13`, a matriz `x` assume as dimensões 2 x 3, e os elementos não referenciados recebem o valor zero, como mostrado na Figura 206.

```
-->x = 7; // matriz 1x1
-->x(2,3) = 13
 x  =
    7.    0.    0.
    0.    0.    13.
```

Figura 206: Expansão de uma matriz

Vetores são matrizes de uma única linha ou de uma única coluna. A Figura 207 mostra a criação na console do Scilab de um vetor linha e de um vetor coluna. Para referenciar o elemento `X(i,1)` de um vetor coluna ou para referenciar o elemento `X(1,i)` de um vetor linha o Scilab permite a utilização da notação `X(i)`, deixando o índice `1` implícito em ambos os casos.

```
-->v = [10 20 30]
 v  =
    10.    20.    30.
-->u = [10; 20; 30]
 u  =
    10.
    20.
    30.
```

Lembrando que o ";" separa linhas

Figura 207: Um vetor linha e um vetor coluna

O Scilab permite que uma parte de uma matriz seja referenciada tanto para a atribuição de valores como para a recuperação de valores armazenados.

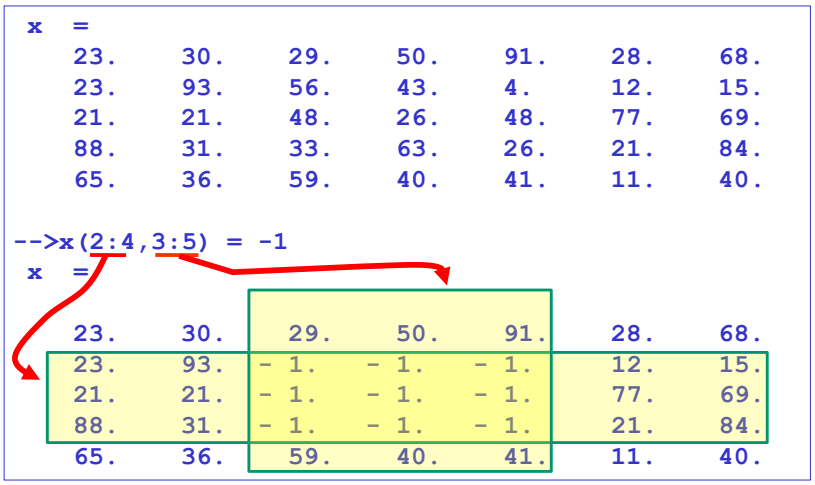


Figura 208: Atribuindo um valor a uma parte de uma matriz

Se `x` contém uma matriz 7x5, `x(2:4,3:5)` denota a parte da matriz compreendida pela interseção das linhas de 2 a 4 e das colunas de 3 a 5, como mostrado na Figura 208.

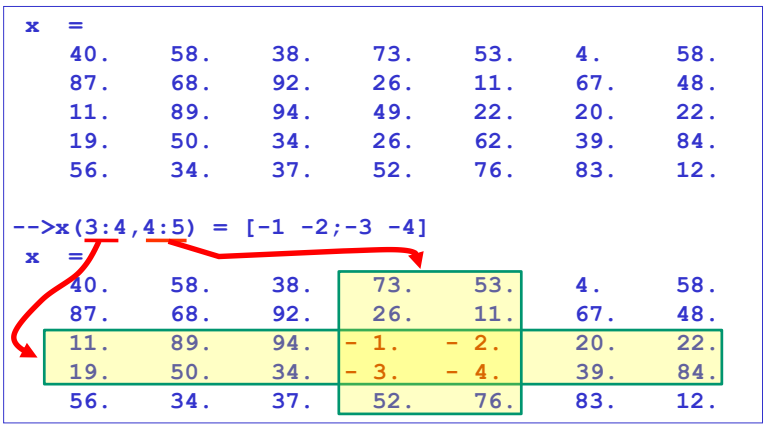


Figura 209: Atribuindo os valores de uma matriz a uma parte de outra matriz

A Figura 209 mostra outro exemplo de utilização desta notação.

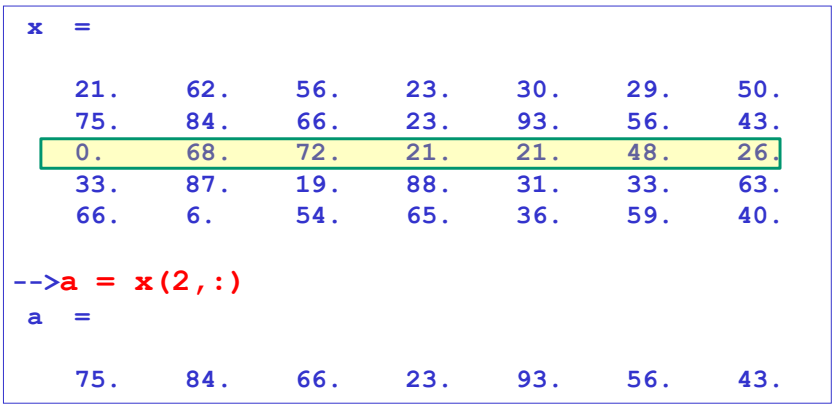


Figura 210: Obtendo todos os valores de uma linha de uma matriz

Ao se referenciar a uma parte de uma matriz que contém, seja todas as linhas, seja todas as colunas da matriz, é possível usar uma notação simplificada com "`:`". A Figura 210 mostra um exemplo de uso desta notação para obter todos os elementos em uma linha de uma matriz.

```
 x  =
    91.    28.    68.    40.    58.    38.    73.
    4.     12.    15.    87.    68.    92.    26.
    48.    77.    69.    11.    89.    94.    49.
    26.    21.    84.    19.    50.    34.    26.
    41.    11.    40.    56.    34.    37.    52.

-->b = x(:,3:5)
 b  =
    68.    40.    58.
    15.    87.    68.
    69.    11.    89.
    84.    19.    50.
    40.    56.    34.
```

Figura 211: Obtendo os elementos de todas as linhas nas colunas de 3 a 5

Outro exemplo está mostrado na Figura 211, onde `x(:,3:5)` designa a parte de `x` formada pelos elementos em todas as linhas e nas colunas de 3 a 5.

6.2 Aritmética matricial

No Scilab as variáveis numéricas são sempre matrizes e, em conseqüência, as operações aritméticas usuais (`+`, `-`, `*`, `/`, `^`) são entendidas pelo Scilab como operações matriciais da álgebra linear. Desta forma, `a*b` designa o produto matricial de uma matriz `a` por uma matriz `b`. Quando o que se deseja é uma operação elemento a elemento, os mesmos símbolos devem ser utilizados precedidos por um "`.`", como `.*` ou `.^`.

```
-->x = [1 2 3; 4 5 6];
-->y = [10 20 30; 40 50 60];
-->x + y
 ans  =
    11.    22.    33.
    44.    55.    66.
-->x - y
 ans  =
  - 9.   - 18.   - 27.
  - 36.  - 45.   - 54.
```

Figura 212: Adição e subtração de matrizes

A Figura 212 mostra exemplos de adição e subtração de matrizes. Como estas operações são sempre feitas elemento a elemento, os operadores `.+` e `.-` não são necessários, e não existem no Scilab.

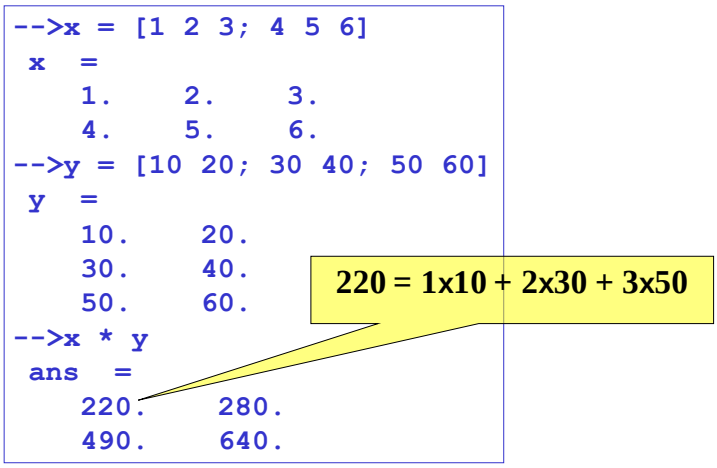


Figura 213: Exemplo de produto matricial

Na Figura 213 nós vemos um exemplo do produto matricial de duas matrizes, obtido com o operador *, e que segue a fórmula da álgebra linear para o produto de uma matriz x de dimensões $m \times n$ por uma matriz y de dimensões $n \times p$, resultando em uma matriz z de dimensões $m \times p$, onde $z_{ij} = \sum_{k=1}^{n} x_{ik} y_{ki}$.

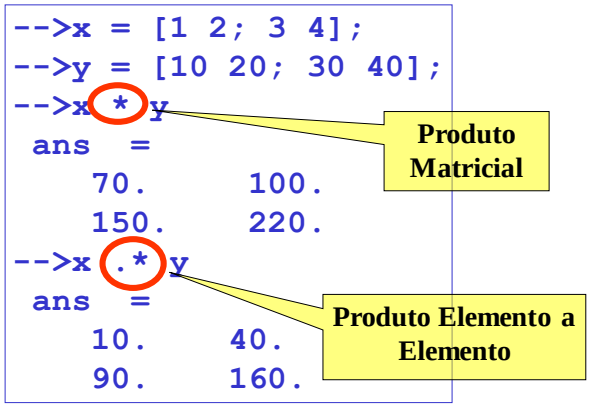


Figura 214: Produto matricial (*) versus produto elemento a elemento (.*) de duas matrizes

O Scilab emite uma mensagem de erro quando ocorre uma tentativa de multiplicação de matrizes com dimensões incompatíveis com a operação. A Figura 214 mostra a diferença entre as operações de produto matricial e produto elemento a elemento.

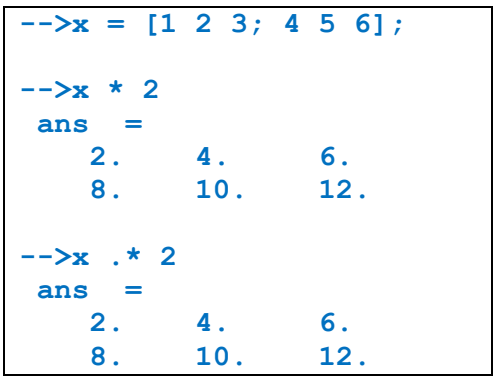


Figura 215: Multiplicando uma matriz por um escalar

Uma matriz pode ser multiplicada por um escalar, caso em que os operadores * e .* são equivalentes, como mostrado na Figura 215.

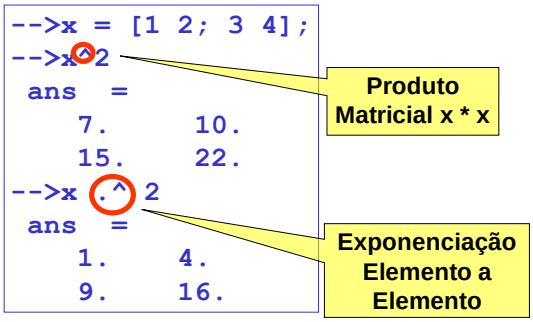


Figura 216: Exponenciação matricial (^) versus exponenciação elemento a elemento (.^)

Quanto à exponenciação, o Scilab interpreta `x^3` como `x*x*x` , ou seja como o produto matricial triplo da matriz `x` por ela mesma. o que só faz sentido quando `x` é uma matriz quadrada. Já `x .^ 3` é interpretado como `x .* x .* x`, ou seja, o produto triplo da matriz `x` por ela mesma, feito elemento a elemento, operação que pode ser feita com matrizes de dimensões arbitrárias. A Figura 216 mostra um exemplo da diferença entre as duas operações.

```
-->a = [1 2 3; 4 5 6]
 a  =
    1.    2.    3.
    4.    5.    6.

-->a'
 ans  =
    1.    4.
    2.    5.
    3.    6.
```

Figura 217: Transpondo uma matriz

Se `a` é uma matriz, `a'` designa a *matriz transposta* de `a`, como mostrado na Figura 217.

```
A  =
    4.    7.    6.
    2.    2.    1.
    1.    1.    6.
-->IA = inv(A)
 IA  =
  - 0.3333333    1.0909091    0.1515152
    0.3333333  - 0.5454545  - 0.2424242
    0.         - 0.0909091    0.1818182
```

Figura 218: A função inv, que produz a matriz inversa

A função `inv` produz a *matriz inversa* da matriz dada como argumento. A Figura 218 mostra um exemplo de sua utilização. Quando multiplicamos uma matriz por sua inversa esperamos obter a matriz identidade, mas não é exatamente isso o que mostra a Figura 219.

```
-->A * IA
 ans  =
    1.           0.   - 4.441D-16
    1.110D-16    1.   - 1.110D-16
    5.551D-17    0.     1.
-->IA * A
 ans  =
    1.     8.327D-17     0.
    0.     1.            0.
    0.     0.            1.
```

Figura 219: O produto de uma matriz por sua inversa calculada pelo Scilab pode ser ligeiramente diferente da matriz identidade

Ali vemos elementos não nulos fora da diagonal principal tanto de **`A * IA`** como de **`IA * A`**. Isso é mais uma manifestação dos erros de arredondamento que ocorrem em operações aritméticas de ponto flutuante. No caso, esses erros não são motivo de preocupação, pois os elementos não nulos fora da diagonal têm valor absoluto ordens de grandeza menores que os elementos das matrizes.

Podemos usar a inversa de uma matriz para resolver um sistema de equações lineares

$$ax = b$$

onde, por exemplo,

$$a = \begin{bmatrix} -2 & -1 & 3 \\ 2 & 1 & 1 \\ -4 & 1 & 3 \end{bmatrix}$$

e

$$b = \begin{bmatrix} 4 \\ 0 \\ 1 \end{bmatrix}$$

Relembrando, podemos resolver a equação multiplicando os dois lados por a^{-1}:

$$a^{-1}ax = Ix = x = a^{-1}b$$

onde I é a matriz identidade. Usando a console do Scilab, o sistema pode ser resolvido com a seqüência de operações mostrada na Figura 220.

```
-->a = [-2 -2 3; 2 1 1;-4 1 3]
 a  =
  - 2.  - 1.    3.
    2.    1.    1.
  - 4.    1.    3.

-->b = [-4 0 1]'
 b  =
  - 4.
    0.
    1.

-->x = inv(a)*b
 x  =
  - 0.5
    2.
  - 1.
```

Figura 220: Resolvendo um sistema de equações lineares

A precisão do resultado pode ser avaliada calculando $ax - b$, o que pode ser feito no Scilab como mostrado na Figura 221.

```
-->residuo = a*x - b
 residuo  =

    0.
  - 2.220D-16
    0.
```

Figura 221: Calculando o erro numérico da solução encontrada

6.3 Construindo matrizes

Vetores com valores regularmente espaçados podem ser construídos como mostrado na Figura 222.

```
-->x = 10:1:13
 x  =
    10.    11.    12.    13.
-->x = 12:-0.5:10
 x  =
    12.    11.5    11.    10.5    10.
-->x = 10:13
 x  =
    10.    11.    12.    13.
```

Figura 222: Construção de vetores regulares

A forma geral é `<valor inicial>:<incremento>:<limite>`. Repare que o incremento pode ser negativo, e que pode ser omitido quando for igual a 1, como no terceiro caso da Figura 222.

Uma outra forma de se conseguir vetores com valores regularmente espaçados é com o uso da função `linspace(<valor inicial>, <valor final>, <número de pontos>)`, onde, além do valor inicial e do valor final, é fornecido o número de pontos em que se deseja dividir o intervalo, ao invés do valor do incremento. A Figura 223 mostra dois exemplos de uso da função `linspace`.

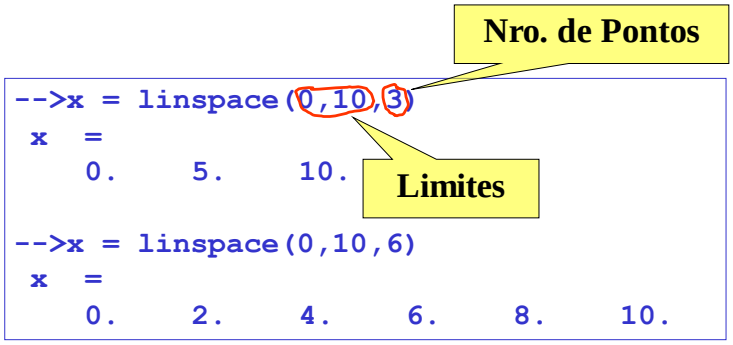


Figura 223: Usando a função `linspace` para construir vetores com valores regularmente espaçados

Para se obter matrizes onde todos os elementos têm o valor 0 ou o valor 1, podem ser utilizadas as funções **`zeros`** e **`ones`**, como mostrado na Figura 224.

```
-->x = zeros(2,3)
 x  =
    0.    0.    0.
    0.    0.    0.
-->y = ones(2,3)
 y  =
    1.    1.    1.
    1.    1.    1.
```

Figura 224: Matrizes com todos os elementos iguais a 0 ou iguais a 1

Outra matriz que se pode obter é a matriz identidade, através da função **`eye`**, como vemos na Figura 225.

```
-->I = eye(4,4)
 I  =
    1.    0.    0.    0.
    0.    1.    0.    0.
    0.    0.    1.    0.
    0.    0.    0.    1.
```

Figura 225: Obtendo uma matriz identidade com a função `eye`

Matrizes com elementos randômicos são muito úteis para programas que fazem simulações de eventos aleatórios, como a chegada de um carro em uma fila. A função **`rand`** gera matrizes onde cada elemento é um número entre 0 e 1, sorteado a cada chamada da função. A Figura 226 mostra dois exemplos de uso desta função.

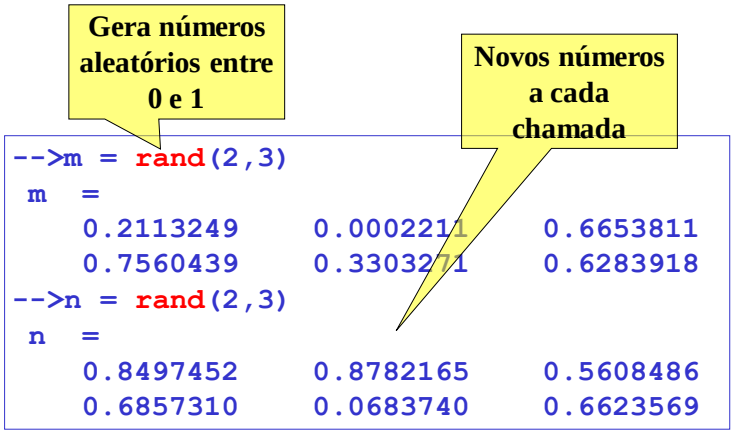


Figura 226: Matrizes randômicas

Algumas vezes é conveniente gerar matrizes aleatórias com valores inteiros entre, digamos, 0 e 100. Isto se faz com um comando como `m = round(rand(2,3)*100)`, muito útil para quem, como o autor destas linhas, necessita com freqüência de exemplos de matrizes com valores inteiros. A função `round` retorna o inteiro mais próximo de seu argumento.

```
-->x = [1 2; 3 4];
-->y = [10 20; 30 40];
-->z = [x y]
 z  =
    1.    2.    10.    20.
    3.    4.    30.    40.
-->z = [x ; y]
 z  =
    1.     2.
    3.     4.
    10.    20.
    30.    40.
```

Figura 227: Construindo matrizes por justaposição de matrizes já existentes

É possível construir matrizes a partir de matrizes já existentes. Se `x` e `y` são matrizes, `[x y]` denota uma nova matriz, com `y` ao lado de `x`, e `[x ; y]` denota uma matriz com `y` abaixo de `x`, como mostrado na Figura 227.

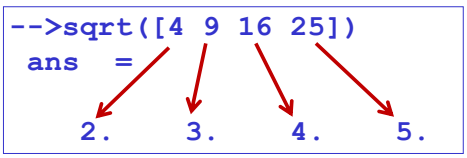


Figura 228: Gerando um vetor como resultado da aplicação de uma função elementar Scilab a um vetor

A Figura 228 mostra outra forma de se construir uma matriz a partir de uma matriz já existente, através da aplicação de uma função elementar do Scilab a uma matriz. A matriz produzida tem as mesmas dimensões da matriz passada como argumento, e cada elemento resulta da aplicação da função ao elemento correspondente da matriz original.

6.4 Matrizes e Gráficos

Matrizes e vetores são imprescindíveis para a construção de gráficos no Scilab. O comando mais simples para a geração de um gráfico é `plot2d(x,y)`, onde `x` e `y` são vetores com o mesmo número de pontos. O Scilab constrói um gráfico unindo por segmentos de reta os pontos `(x(1), y(1))`, `(x(2),y(2))` , e assim por diante até o último par de pontos.

```
-->x = [1 4 7 11]; y = [12 6 15 7];

-->plot2d(x,y)
```

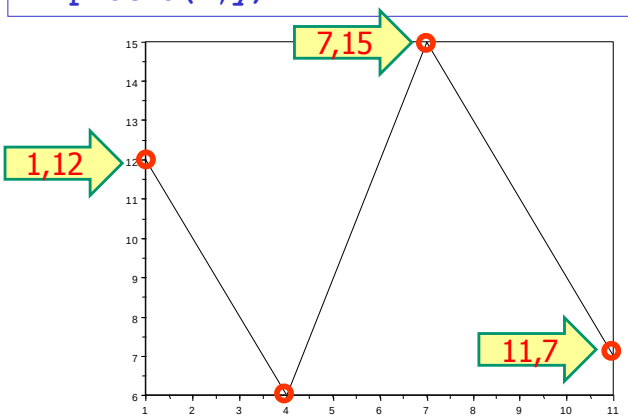


Figura 229: Exemplo de gráfico obtido com plot2d

A Figura 229 mostra um exemplo de gráfico obtido com o comando **plot2d**; outro exemplo está na Figura 230, que mostra que poligonais arbitrárias podem ser traçadas com **plot2d**.

Uma infinidade de parâmetros pode ser utilizada no comando **plot2d**, determinando cores e espessuras de linhas, tracejados, escalas, etc. Neste curso nós veremos apenas comandos básicos; você pode usar o help do Scilab para saber mais e obter um gráfico com um acabamento melhor.

```
-->x = [2 5 3 4]; y = [ 3 1 4 7];

-->plot2d(x,y)
```

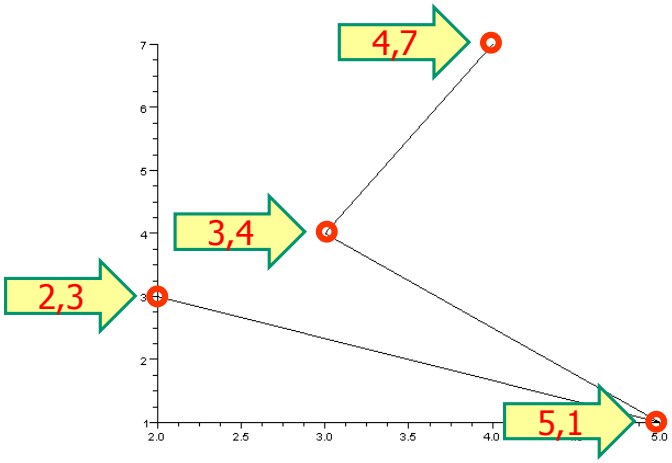


Figura 230: Outro exemplo de gráfico obtido com plot2d

Para se obter um gráfico da função seno, podemos fazer como mostrado na Figura 231. Primeiramente é gerado o vetor **x** de abscissas e, depois, o vetor **y** é obtido pela aplicação da função **sin** ao vetor **x**. O comando **plot2d(x,y)** gera o gráfico.

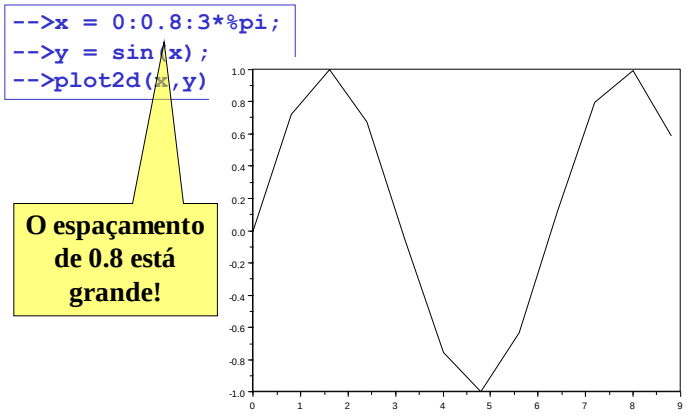


Figura 231: Gráfico da função seno com espaçamento excessivo

A “curva” obtida está toda quebrada, com cotovelos visíveis. É que, ao construirmos um gráfico a partir de valores em vetores, estamos fazendo uma transformação digital-analógica adequada para consumo humano. No caso, o número de pontos utilizado para as abscissas foi pequeno para a nossa acuidade visual. Um resultado melhor (e que mostra que a geração de um vetor pela função **`linspace`** é mais confortável nessas ocasiões) está mostrado na Figura 232.

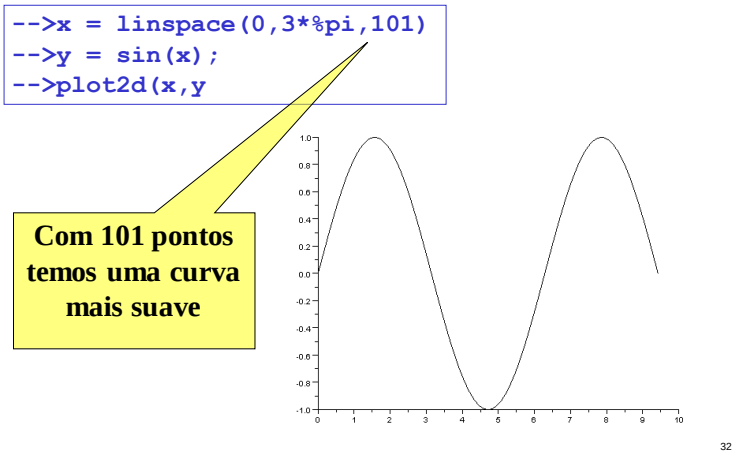


Figura 232: Gráfico da função seno com um espaçamento agradável

A função **`plot2d`** pode ser usada para traçar várias curvas em um único gráfico. O comando **`plot2d(x,M)`** , onde `x` é um vetor coluna, e **`M`** é uma matriz com o mesmo número de linhas de **`x`** faz um gráfico de **`x`** versus cada coluna de **`M`**.

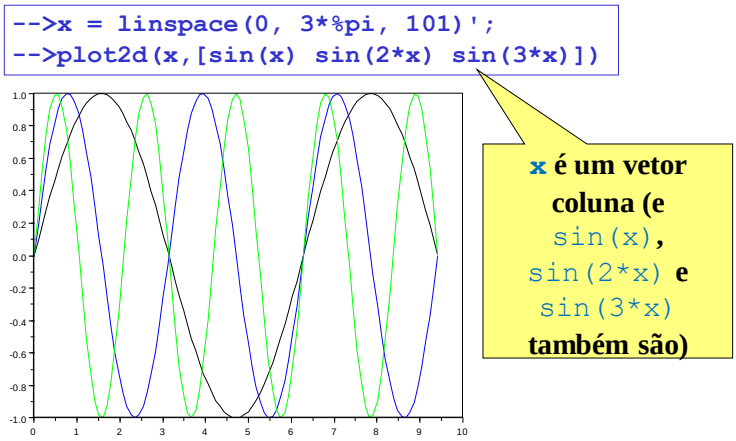


Figura 233: Gráfico com várias curvas obtido com plot2d

A Figura 233 mostra um gráfico obtido dessa forma. Repare que o vetor `x` é um vetor coluna, e que, como na Figura 227, a matriz cujas colunas são mostradas no gráfico é construída por justaposição dos vetores coluna `sin(x)` , `sin(2*x)` e `sin(3*x)`.

6.5 Gráficos 3D

Gráficos tridimensionais podem ser produzidos no Scilab de muitas maneiras; o comando mais básico é o `plot3d`. Em sua forma mais simples, é preciso gerar um vetor `x`, outro vetor `y`, e um valor para `z` para cada par `(x(i),y(i))`. Por exemplo, a execução do programa da Figura 234 gera o gráfico da Figura 235.

```
x = linspace(-2*%pi,2*%pi,90);
y = x;
for i=1:length(x)
  for j=1:length(y)
    z(i,j) = sin(x(i))*cos(y(j));
  end
end
plot3d(x,y,z)
```

Figura 234: Exemplo de uso de plot3d

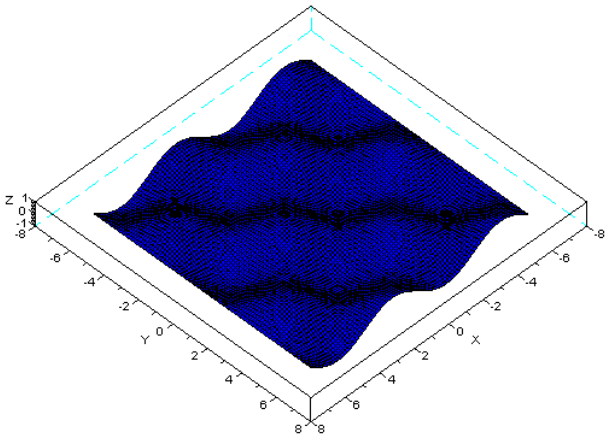


Figura 235: $f(x,y) = seno(x)cos(y)$

Se substituirmos a função por `z(i,j) = 10 -(x(i)^2 + y(j)^2)`, por exemplo, vamos obter o gráfico da Figura 236.

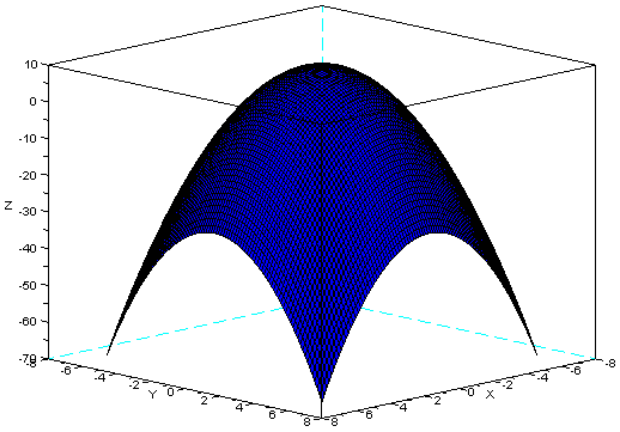


Figura 236: $f(x,y) = 10 - (x^2 + y^2)$

Gráficos mais coloridos são obtidos com a função **plot3d1**, que permite também a alteração do mapa de cores. Veja o exemplo do programa da Figura 237, que você não precisa entender em detalhes para modificá-lo conforme seu interesse.

```
clear();
x = linspace(-2*%pi,2*%pi,60);
y = x;
for i=1:length(x)
  for j=1:length(y)
    z(i,j) = sin(x(i))*cos(y(j));
  end
end
curFig = scf(3);
plot3d1(x,y,z);
cmap = curFig.color_map;
curFig.color_map = autumncolormap(64);
xtitle("f(x,y) = seno(x).cos(y)","Eixo x","Eixo y","Eixo z")
```

Figura 237: Exemplo de uso de plot3d1 com mapa de cores

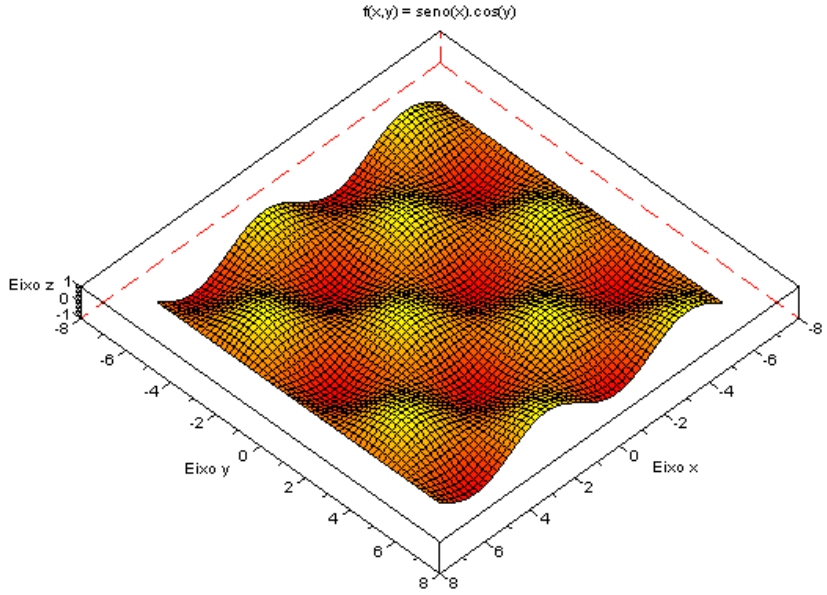


Figura 238: $f(x,y) = seno(x)cos(y)$ com plot3d1 e mapa de cores

Além do `autumncolormap`, o Scilab oferece vários outros mapas que associam valores a cores. Para descobri-los, bata `help colormap` na console do Scilab.

A janela gráfica permite também que você gire o gráfico com o mouse, conseguindo outras visões de seus dados. O gráfico abaixo foi obtido utilizando este recurso de rotação.

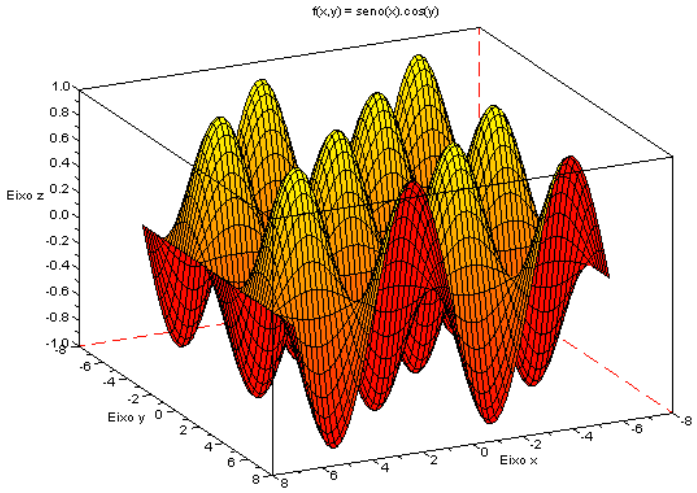


Figura 239: Rotação do gráfico da Figura 238

Estes são exemplos simples mas que podem ser úteis em cálculo e outras disciplinas. Existem infinitas combinações de muitos e muitos parâmetros que alteram a forma dos gráficos, que você pode pesquisar e experimentar no Scilab.

6.6 Matrizes de Strings

Matrizes podem ter strings como valores de seus elementos, como mostram os exemplos na Figura 240.

```
-->a = ["s1" "s2"]
 a  =
!s1  s2  !
-->b = ["s1" ; "s2"]
 b  =
!s1  !
!    !
!s2  !
```

Figura 240: Exemplos de matrizes de strings

Diversas funções oferecidas pelo Scilab têm vetores de strings como resultado, como a função `tokens`, utilizada para obter partes de um strings separadas por um delimitador. Um exemplo de uso desta função está na Figura 241.

```
-->tokens("C:\Users\Osvaldo\Dropbox\Ensino\dcc001\2012-2\Módulo 06","\")
 ans  =

!C:          !
!            !
!Users       !
!            !
!Osvaldo     !
!            !
!Dropbox     !
!            !
!Ensino      !
!            !
!dcc001      !
!            !
!2012-2      !
!            !
!Módulo 06   !
```

Figura 241: Exemplo de uso da função tokens com delimitador "\"

Quando o caractere delimitador dos tokens é o espaço em branco, ele pode ser omitido na chamada, como mostra a Figura 242.

```
-->t = tokens('Ouviram do Ipiranga as margens plácidas")
 t  =

!Ouviram     !
!            !
!do          !
!            !
!Ipiranga    !
!            !
!as          !
!            !
!margens     !
!            !
!plácidas    !
```

Figura 242: Exemplo de uso da função `tokens` usando espaço como separador

O ligeiro desalinhamento presente nos dois exemplos se deve a um bug do Scilab, que ocorre na exibição de strings com caracteres acentuados.

6.7 Matrizes e Expressões Lógicas

O resultado de uma expressão relacional envolvendo matrizes é uma matriz de valores booleanos que resultam da expressão aplicada elemento a elemento das matrizes, como mostram os exemplos na Figura 243.

```
-->a = [3 7;8 2]
 a  =
    3.    7.
    8.    2.
-->a > 5
 ans  =
  F T
  T F
```

```
-->a = [3 7; 8 2];

-->b = [5 6; 7 8];

-->a > b
 ans  =
  F T
  T F
```

Figura 243: Expressões relacionais gerando matrizes booleanas

Uma expressão relacional envolvendo matrizes pode ser empregada em um comando **if**, mas isso deve ser feito com muito cuidado, pois a cláusula **then** só será executada se *todos* os elementos da matriz booleana resultante forem iguais a **%t**. A Figura 244 mostra um exemplo, onde somente o segundo **if** tem a sua cláusula **then** executada, pois a matriz resultante da comparação **a > 0** tem todos os elementos iguais a **%t**.

```
-->a = [3 9; 12 1]
-->x = 0; y = 0;
-->if a > 5 then; x = 10000; end;
-->if a > 0 then; y = 10000; end;
-->[x y]
 ans  =
    0.    10000.
```

Figura 244: Exemplos de emprego de expressões relacionais matriciais em comandos if

Se **A** for uma matriz e **MB** uma matriz booleana com as mesmas dimensões de **A**, **A(MB)** designa aqueles elementos de **A** com correspondentes em **MB** iguais a **%t**.

```
-->A = [3 9; 12 1]
 A  =
    3.    9.
    12.   1.

-->MB = [%t %f; %f %t]
 MB  =
  T F
  F T

-->A(MB) = 0
 A  =
    0.    9.
    12.   0.
```

Figura 245: Uso de matriz booleana para seleção de elementos de uma matriz

Isso nos permite selecionar elementos de uma forma elegante, como mostra a Figura 246

```
-->a = [3 9; 12 1];
-->a(a>5) = -1
 a  =
    3.   - 1.
  - 1.     1.
```

Figura 246: Exemplo de seleção de elementos de uma matriz por uma matriz booleana

7 Loops

Loops são construções que você pode colocar em um programa para prescrever comportamentos repetitivos em sua execução. Loops são a essência da programação: nós, programadores, trabalhamos pouco para construí-los; computadores podem trabalhar muito para executá-los. Veremos neste módulo dois comandos para a programação de loops, os comandos `while` e `for` que, com algumas variações, existem em praticamente todas as linguagens de programação.

7.1 O comando `while`

Como um primeiro exemplo vamos fazer um programa que, repetidamente, leia coeficientes de equações de segundo grau e calcule suas raízes. A cada equação processada, o programa deve interrogar o usuário sobre seu desejo de resolver mais uma equação. O processamento de cada equação deve ser igual ao produzido pelo programa da Figura 195.

Um comportamento repetitivo pode ser obtido com o uso do comando `while`, que tem a seguinte forma:

```
while <condição>
  <bloco de comandos>
end
```

Sua execução resulta em repetidas execuções dos comandos no `<bloco de comandos>` (que chamamos de *corpo do loop*), repetição que só se interrompe quando a expressão lógica `<condição>` não se verificar.

Para transformarmos o programa não repetitivo em repetitivo, muitas vezes é possível seguir um padrão como mostrado na Figura 247.

```
continua = %T;
while continua
  // Processamento de um item
  ...
  // Decisão
  decisao = ...
  input('Continua? (s/n)','string');
  continua = decisao == 's';
end
printf('\nObrigado!');
```

Figura 247: Padrão de programa com loop controlado por uma ação do usuário

Aqui a `<condição>` do `while` é uma expressão lógica dada pelo valor de uma única variável, a que demos o nome de `continua`. Seu valor inicial é `%t`, o que garante uma primeira execução do corpo do loop. Ao fim de cada execução do corpo do loop, o usuário deve digitar `'s'` se desejar processar outro item, e `'n'` senão (na verdade, qualquer string diferente de `'s'` serviria).

```
// Cálculo das raízes de diversas equações
// de 2o grau
continua = %t;
while continua
  // Processamento de uma equação
  a = input("Digite o valor de a:");
  b = input("Digite o valor de b:");
  c = input("Digite o valor de c:");

  delta = b^2 - 4*a*c
  if delta >= 0 then
    r1 = (-b+sqrt(delta))/(2*a);
    r2 = (-b-sqrt(delta))/(2*a);
    printf("As raízes são %g e %g",r1,r2);
  else
    printf("As raízes são complexas");
  end
  // Decisão de continuação pelo usuário
  decisao = input("Outra equação? (s/n)","string");
  continua = decisao == "s";
end
printf("Obrigado, e volte sempre.")
```

Figura 248: Programa para cálculo das raízes de diversas equações de 2o grau

O emprego do comando **while** deve ser feito com atenção, pois você pode prescrever um loop que nunca irá parar, como no exemplo da Figura 249. O **while** só seria interrompido quando **x** for maior que 10, o que nunca acontecerá porque **x** vale 0 inicialmente e, a cada passo, fica ainda menor (no Scilab você pode interromper a execução de um programa que aparenta estar em loop infinito através do menu *Controle/Abortar*).

```
x = 0;
while x <= 10
  printf("\nx = %g", x)
  x = x - 0.2;
end
```

Figura 249: Um loop infinito

7.2 Loop para o Cálculo de Fatorial

Como um segundo exemplo de uso do comando **while**, vamos atacar o problema do cálculo do fatorial de um número a ser lido em tempo de execução. O Scilab oferece diretamente a função **factorial(n)** que já faz este cálculo, mas aqui nós vamos mostrar como calcular um fatorial fazendo uma multiplicação de cada vez, o que provavelmente é feito pela implementação da função **factorial**.

Nós sabemos que $n! = 1 \times 2 \times \ldots \times n = \prod_{k=1}^{n} k$, e que portanto teremos que realizar repetidas multiplicações para obter o fatorial. Sabemos também que $0! = 1$, e que $k! = (k-1)! \times k$.

```
k = 1; fat = 1; // fat == (k-1)!
while k <= n
    fat = fat * k;
    //== (k-1)! * k == k!
    k = k + 1;
    // fat == (k-1)!
end
// fat == (k-1)! e k == n+1
// portanto, fat == n!
```

Figura 250: Cálculo de fatorial utilizando o comando `while`

Já temos condições de compreender o código mostrado na Figura 250, que utiliza duas variáveis, `fat` e `k`. O objetivo é obter `fat` tal que `fat == n!`. O raciocínio empregado se baseia essencialmente no fato de que, como é possível obter o fatorial de `k` multiplicando por `k` o fatorial de `k-1`, e como o fatorial de 0 é igual a 1, podemos obter o fatorial de `n` por multiplicações sucessivas de 1 por 1, 2, ..., `n`:

- Como preparação para a execução do loop, as variáveis `k` e `fat` são inicializadas com o valor 1.
- O loop é construído explorando a propriedade `fat == (k-1)!`, que é estabelecida pela preparação do loop (dado que `k` == 1, e (`k`-1)! == 0! == 1), e restabelecida ao término de cada execução do corpo do loop.
- A variável `k` é incrementada de 1 a cada passagem do loop.
- Quando o loop termina, temos `k == n + 1`, e como `fat == (k-1)!`, temos em `fat` o valor de `n!` , como queríamos.

Em um loop, uma propriedade que é válida ao início e ao término de cada execução de seu corpo é chamada de *invariante do loop*. Na Figura 250 a propriedade `fat == (k-1)!` é um invariante do loop. Um exemplo de execução deste loop está na Figura 251.

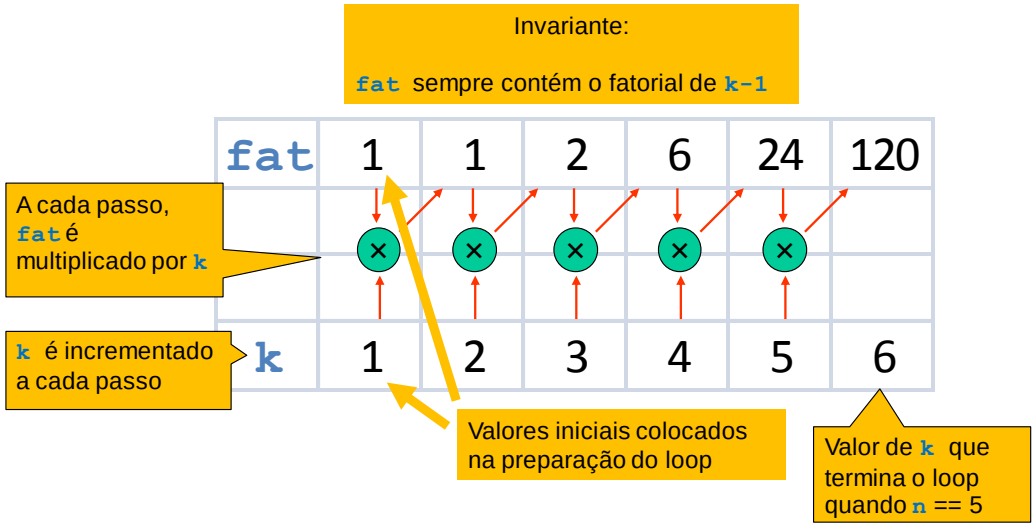


Figura 251: Exemplo de cálculo do fatorial usando o loop da Figura 250

7.3 Loop para o Cálculo da Soma dos Elementos de um Vetor

Como mais um exemplo, vamos explorar invariantes de loop para obter em uma variável `s` a soma de todos os elementos de um vetor `A` com `n` elementos. No Scilab, isso pode ser feito diretamente pelo comando `s = sum(A)`; a Figura 252 mostra como fazê-lo utilizando um loop, fazendo uma soma de cada vez, como provavelmente é implementada a função `sum`.

```
s = 0; k = 1;
// s == sum(A(1:k-1)),
// lembrando que A(1:0) == []
while k <= n
  s = s + A(k);
  // s == sum(A(1:k))
  k = k + 1;
  // s == sum(A(1:k-1))
end
// s == sum(A(1:k-1)) e k == n+1
// portanto, s == sum(A(1:n))
```

Figura 252: Loop para soma dos elementos de um vetor

O objetivo é obter `s` tal que `s == sum(A)`, e o raciocínio empregado na construção do loop é o seguinte:

- Como preparação para o loop, fazemos `s = 0` e `k = 1`;
- O loop explora o invariante `s == sum(A(1:k-1))`, que é estabelecido pela preparação do loop (lembrando que `A(1:0) == []`, a matriz vazia), e restabelecido ao término de cada execução do corpo do loop;
- A variável `k` é incrementada de 1 a cada passagem do loop;
- Quando o loop termina, `k == n + 1`, e como `s == sum(A(1:k-1))`, temos `s == sum(A(1:n))`, como queríamos.

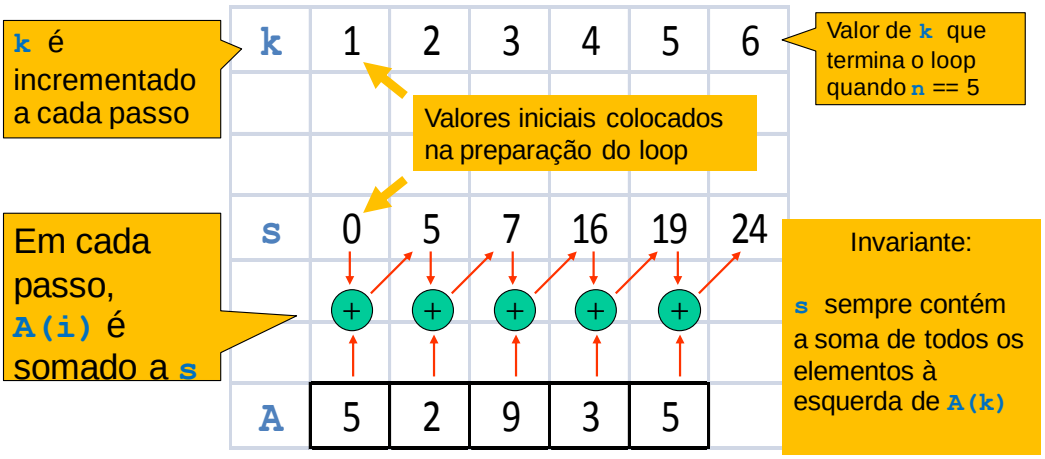

k é incrementado a cada passo
k 1 2 3 4 5 6
Valor de k que termina o loop quando n == 5
Valores iniciais colocados na preparação do loop
s 0 5 7 16 19 24
Em cada passo, A(i) é somado a s
Invariante:
s sempre contém a soma de todos os elementos à esquerda de A(k)
A 5 2 9 3 5

Figura 253: Exemplo de cálculo da soma dos elementos de um vetor pelo loop da Figura 252

7.4 Loop para encontrar o Menor Valor presente em um Vetor

Como mais um exemplo, vamos mostrar um loop que atribui a uma variável `m` o menor valor presente em um vetor `A` de `n` elementos, o que poderia ser feito no Scilab com o comando `m = min(A)`.

```
m = A(1); k = 2; // m == min(A(1:k-1))
while k <= n
    if A(k) < m then
        m = A(k);
    end
    // m == min(A(1:k))
    k = k + 1;
    // m == min(A(1:k-1))
```

```
end
// m == min(A(1:k-1)) e k == n + 1;
// portanto, m = min(A(1:n))
```

Figura 254: Loop para encontrar o menor valor presente em um vetor

Nosso objetivo é obter `m` tal que `m == min(A)`; a lógica empregada na construção do loop é a seguinte:

- Como preparação para o loop, fazemos `m = A(1)` e `k = 2`;
- O loop explora o invariante `m == min(A(1:k-1))`, que é estabelecido pela preparação do loop (pois `A(1) == min(A(1:1))`), e restabelecido ao término de cada execução do corpo do loop;
- A variável `k` é incrementada de 1 a cada passagem do loop;
- Quando o loop termina, `k == n + 1`, e como `m == min(A(1:k-1))`, temos `m == min(A(1:n))`, como queríamos.

Um exemplo de execução deste loop está na Figura 255.

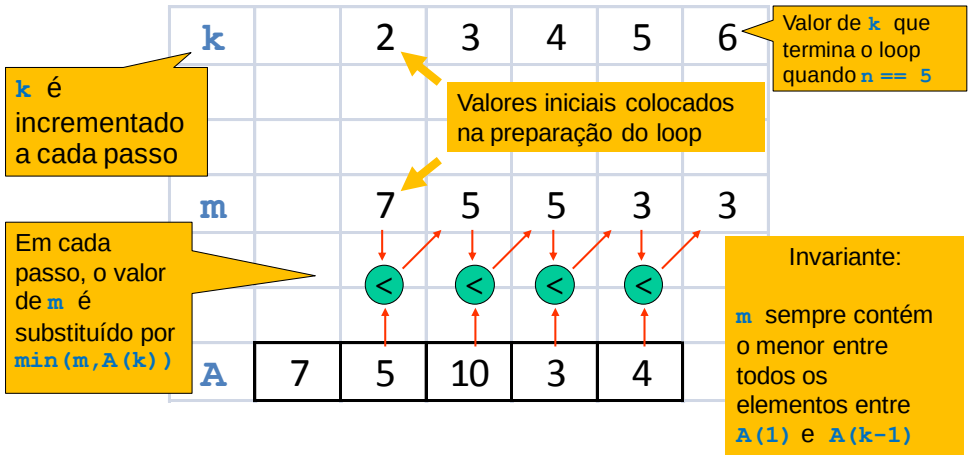


Figura 255: Exemplo de determinação do menor valor presente em um vetor usando o loop da Figura 254

7.5 Loop para encontrar o Máximo Divisor Comum usando o Algoritmo de Euclides

Como ainda outro exemplo de uso do comando **`while`**, vamos apresentar o algoritmo de Euclides, proposto em ... 300 A.C. e que é utilizado até hoje[45]! Este algoritmo encontra o máximo divisor comum de dois inteiros positivos. Por definição, $mdc(m,n)$ onde m e n são inteiros positivos, é o maior dentre todos os divisores comuns a m e n. O algoritmo se baseia no fato de que se substituirmos o maior dentre m e n pela diferença entre m e n, o máximo divisor comum não se altera.

A prova desta propriedade não é difícil. Queremos mostrar que se $d = mdc(m,n)$ e $m > n$, então $d = mdc((m-n),n)$.

Parte 1: Se $d = mdc(m,n)$ então d também divide $m - n$

Se d é um divisor de m, então

$$m = d.x$$

onde $x \geq 1$ é um inteiro. Mas d é também um divisor de n, e portanto

$$n = d.y$$

[45] "Euclidean algorithm - Wikipedia, the free encyclopedia", acessado 8 de março de 2011, http://en.wikipedia.org/wiki/Euclidean_algorithm#cite_note-2.

onde $y \geq 1$ é um inteiro. Temos então

$$m - n = d(x - y)$$

supondo que $m > n$, ou $x > y$. Ou seja, se d é um divisor de m e de n, m é também um divisor de $m - n$.

Parte 2: Se $d = mdc(m,n)$ então d é o maior dentre os divisores de $(m - n)$ e n.

Suponhamos que exista $D > d$ tal que $(m - n) = D.z$ e $n = D.w$; teremos

$$m - D.w = D.z$$

ou

$$m = D.(w + z)$$

D seria portanto divisor de m e de n e, como $D > d$ por hipótese, e seria maior que $mdc(m,n)$, o que contradiz a definição de máximo divisor comum

■

Sabemos também que $mdc(a,a) = a$, para qualquer a inteiro positivo. Com isso podemos construir o programa da Figura 256.

```
m = input("m = "); n = input("n = ");
a = m; b = n;
// a, b inteiros; a > 0, b > 0;
// mdc(a,b) == mdc(m,n)
while a ~= b
    if a > b then
        a = a - b;
    else
        b = b - a;
    end
    // mdc(a,b) == mdc(m,n)
    // a, b inteiros; a > 0, b > 0;
end
// mdc(a,b) == mdc(m,n)
// a, b inteiros; a > 0, b > 0;
// Como a == b, mdc(a,b) == a == b == mdc(m,n)
printf("MDC(%d,%d) = %d",m,n,a)
```

Figura 256: Programa para cálculo do máximo divisor comum pelo algoritmo de Euclides

A construção deste loop emprega a seguinte lógica:

- na preparação do loop os comandos `a = m` e `b = n`, estabelecem trivialmente o invariante do loop, `mdc(a,b) == mdc(m,n)`, `a > 0` e `b > 0` (supondo que `m > 0` e `n > 0`)
- a cada passo, se `a` for maior que `b`, seu valor é substituído por `a - b`, e se `b` for maior do que `a`, seu valor é substituído por `b - a`. Nos dois casos o invariante é mantido, em vista do que demonstramos acima e do fato de estarmos subtraindo de um número um valor positivo menor que ele;
- quando o loop termina, temos `a == b`, e portanto `a` (ou `b`, claro) contém o valor desejado do `mdc(m,n)`.

A Figura 257 mostra um exemplo de execução deste loop.

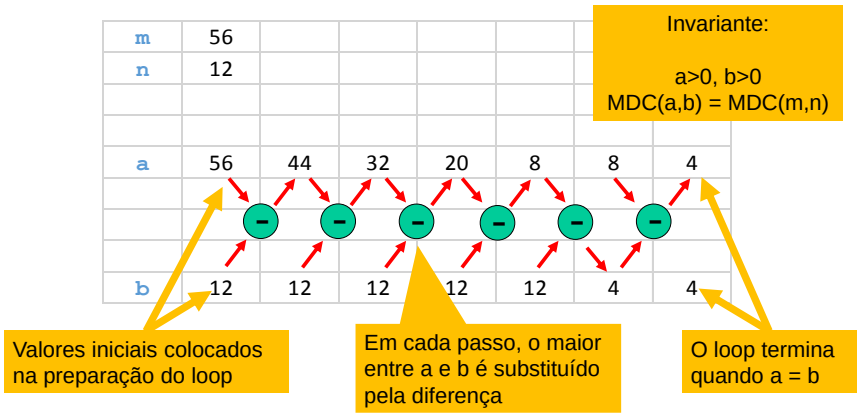


Figura 257: Exemplo de loop para cálculo do máximo divisor comum

7.6 O comando `for`

Loops como o da Figura 250 são tão freqüentes em programação que existe um outro comando, o **`for`**, que substitui o comando **`while`** com vantagens em alguns casos. Na Figura 258 nós vemos um programa que também calcula o fatorial de um número, utilizando o comando **`for`**.

```
// Cálculo do fatorial de um número

// Leitura do número
n = input("n = ");

// Cálculo do fatorial
fat = 1;
for k = 1:n
    fat = fat * k;
end

// Impressão do resultado
printf("O fatorial de %d é %d",n,fat)
```

Figura 258: Cálculo de fatorial utilizando o comando `for`

A simplificação com relação à versão com **`while`** vem da inicialização da variável **`k`** com o valor 1 e da condição de parada, implícitas no comando **`for k = 1:n`**. Também implícito neste **`for`** está o comando **`k = k + 1`**.

A forma geral de um comando **`for`** está mostrada na Figura 259.

```
for <variável> = <vetor>
  <bloco for>;
End
```

Figura 259: Forma geral de um comando for

Aqui **`<vetor>`** contém os valores atribuídos à *variável indexadora* **`<variável>`** a cada iteração do loop.

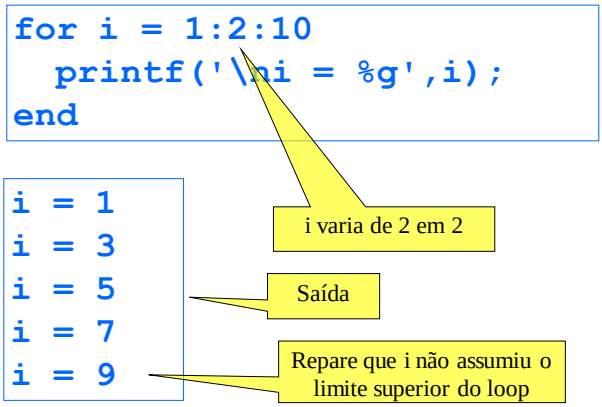


Figura 260: Exemplo de comando for com passo diferente de 1

No exemplo da Figura 260,

- A variável indexadora é `i`;
- O vetor de valores `1:2:10` é igual a `[1 3 5 7 9]`; são estes os valores que `i` assumirá a cada execução do corpo do loop.
- O "`\n`" na frase do comando `printf` é uma *sequência de escape*, isto é, uma codificação em caracteres imprimíveis de um caracter não imprimível – no caso, *Line Feed*, que produz uma nova linha na saída.

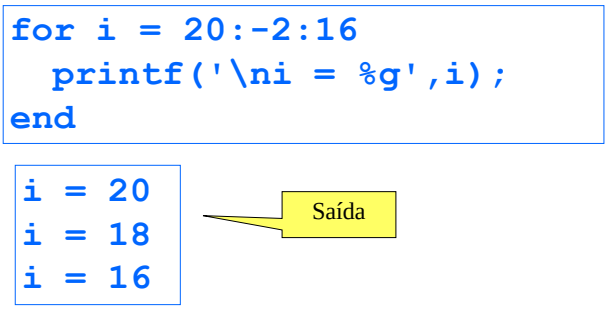


Figura 261: Exemplo de for com passo negativo

O passo de um `for` pode ser negativo, como mostrado na Figura 261, e a variável de controle pode assumir valores não inteiros, como na Figura 262.

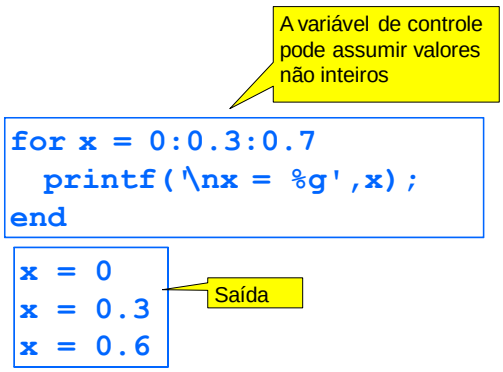


Figura 262: Exemplo de for com a variável de controle assumindo valores não inteiros

7.7 Geração e Impressão de Tabelas

x	seno(x)
0.0	0.0000
0.2	0.1987
0.4	0.3894
0.6	0.5646
0.8	0.8415

Figura 263: Tabela de Senos

Vamos agora usar o comando **for** para a construção de uma tabela como a da Figura 263 , com **x** variando de 0 a 2π, de 0.2 em 0.2, e vamos aproveitar a oportunidade para aprender mais sobre o controle de saída com o comando **printf**.

```
// Tabela da função Seno
for x = 0:0.2:2*%pi
  printf("%g %g",x, sin(x))
end
```

Figura 264: O programa Tabela_de_Senos_1.sce

O programa da Figura 264 parece atender à especificação, mas quando executado pelo Scilab produz uma saída de difícil compreensão, como mostrado na Figura 265.

```
-->
 0 00.2 0.1986690.4 0.3894180.6 0.5646420.8 0.7173561 0.841471
```

Figura 265: Primeiros caracteres da saída do programa Tabela_de_Senos_1.sce

Um primeiro problema a corrigir é a separação em linhas, o que pode ser obtido usando a sequência de escape "**\n**" na frase do comando **printf**. Com isso nós chegamos ao programa Tabela_de_Senos_2.sce, mostrado na Figura 266.

```
// Tabela da função Seno
for x = 0:0.2:2*%pi
  printf("\n%g %g",x, sin(x))
end
```

Figura 266: O programa Tabela_de_Senos_2.sceFigura 263

A saída do Tabela_de_Senos_2.sce, cujas primeiras linhas estão mostradas na Figura 267, melhorou, mas ainda não está satisfatória.

```
0 0
0.2 0.198669
0.4 0.389418
0.6 0.564642
0.8 0.717356
1 0.841471
1.2 0.932039
```

Figura 267: Saída do programa Tabela_de_Senos_2.sce

Os problemas de alinhamento são causados pelo uso do código de formato **%g**, que não especifica o número de colunas que um número irá ocupar. O número de colunas é igual ao número de caracteres com a fonte tipográfica não proporcional usada pelo Scilab, onde todas as letras ocupam o mesmo espaço. Com uma fonte proporcional, como esta que você está

lendo, um i ocupa um espaço menor do que um m, o que é bem visível quando comparamos iiii com mmmm.

```
// Tabela da função Seno

// Impressão do cabeçalho
printf("\n x   seno(x)")

// Impressão das linhas da tabela
for x = 0:0.2:2*%pi
  printf("\n%3.1f %7.4f",x, sin(x))
end
```

Figura 268: O programa Tabela_de_Senos_3.sce

A Figura 268 mostra o programa Tabela_de_Senos_3.sce que produz a saída mostrada parcialmente na Figura 269.

```
 x   seno(x)
0.0  0.0000
0.2  0.1987
0.4  0.3894
0.6  0.5646
0.8  0.7174
1.0  0.8415
1.2  0.9320
```

Figura 269: Primeiras linhas da saída do programa Tabela_de_Senos_3.sce

Nessa última versão,

- É impressa uma linha com cabeçalhos para a tabela
- Para a formatação de `x` é usado o código `%3.1f`, que especifica um campo ocupando 3 colunas ao todo na impressão, com 1 casa decimal;
- Para a formatação de `sin(x)`, o código `%7.4f` especifica um campo com um total de 7 colunas, com 4 casas decimais.

Todo comando `for` pode ser substituído por um comando `while`; a Figura 270 mostra outro exemplo dessa substituição. O inverso não é verdadeiro: nem todo loop com `while` pode ser substituído com facilidade por um loop com o comando `for`, como é o caso do algoritmo de Euclides.

```
for x = 0:0.2:2*%pi
  printf("\n%3.1f %7.4f",...
         x, sin(x))
end
```

```
x = 0;
while x <= 2*%pi
  printf("\n%3.1f %7.4f",...
         x, sin(x))
  x = x+0.2;
end
```

Figura 270: Mesmo loop obtido com comandos for e while

7.8 Comandos Aninhados

Blocos de comandos definidos por comandos como **`if`** e **`for`** podem conter qualquer tipo de comando, incluindo comandos de atribuição, de entrada e/ou saída, mas também outros **`if`**s e outros **`for`**. Este aninhamento de comandos proporciona uma grande flexibilidade para o programador.

Para ilustrar o uso de **`if`**s aninhados vamos agora desenvolver um programa que:

- Leia o nome do aluno, que responde, por exemplo, "*Paulo*";
- Leia também o total de pontos obtidos pelo aluno;
- Imprima, conforme o caso, a frase *<aluno>, com <pontos> você obteve o conceito X!*, onde X é determinado pela Tabela 3. Exemplos seriam *Paulo, com 81 pontos você obteve o conceitoB!*, ou *Paulo, com 90 pontos você obteve o conceito A!*

Tabela 3: Pontos e Conceitos

Pontos	Conceito
$90 \leq Pontos \leq 100$	A
$80 \leq Pontos < 90$	B
$70 \leq Pontos < 80$	C
$60 \leq Pontos < 70$	D
$40 \leq Pontos < 60$	E
$0 \leq Pontos < 40$	F

O programa PontosConceito.sce, apresentado na Figura 271, atende a esta especificação, acrescida da suposição de que o usuário irá digitar sempre valores entre 0 e 100.

Neste programa você deve observar que:

- A leitura do nome e dos pontos do aluno é idêntica à utilizada no programa PassouNaoPassou.sce;
- Se o <bloco "senão"> do primeiro **`if`** chegar a ser executado, é porque temos **`Pontos < 90`**, e por isto basta verificar que **`Pontos >= 80`** para concluir que o conceito é B;
- Se o <bloco "senão"> do segundo **`if`** chegar a ser executado, é porque temos **`Pontos < 80`**, e por isso basta verificar que **`Pontos >= 70`** para concluir que o conceito é C;
- O mesmo raciocínio se aplica aos outros **`if`**s, até o mais interno, onde o <bloco "senão"> não necessita de mais testes para concluir que o conceito é F.
- Detalhe, mas muito útil: o **`printf`** no fim do programa ilustra o uso de **`...`** para indicar ao Scilab que um comando se prolonga por mais uma linha.

```
// Leitura do nome do aluno
Nome = input("Seu nome, por favor:");

// Leitura dos pontos obtidos
Pontos = input(Nome + ", quantos pontos você conseguiu?");

// Determinação do conceito
if Pontos >= 90 then
  Conceito = "A";
else
  if Pontos >= 80 then
    Conceito = "B";
  else
    if Pontos >= 70 then
      Conceito = "C";
    else
      if Pontos >= 60 then
        Conceito = "D";
      else
        if Pontos >= 40 then
          Conceito = "E"
        else
          Conceito = "F"
        end
      end
    end
  end
end
printf("%s, com %g pontos você obteve o conceito %s!",...
       Nome, Pontos, Conceito)
```

Figura 271: O programa PontosConceito.sce

Como já dissemos e continuaremos a dizer ao longo desse curso, programas são feitos para serem executados por computadores, mas também para serem lidos por humanos. Ao fazer um programa você deve se preocupar com a sua *legibilidade*, tanto por outras pessoas que venham a trabalhar com ele, como com você mesmo, algum tempo depois de tê-lo escrito.

A disposição gráfica dos comandos em um programa tem grande influência em sua legibilidade.Talvez você já tenha reparado que o próprio editor SciNotes faz com que os blocos nos comandos `if` e `for` sempre apareçam ligeiramente deslocados para a direita, com relação à posição do comando que os delimita. Isso é porque, com esse deslocamento, a identificação do começo e do fim de cada bloco fica muito facilitada, e os possíveis fluxos de execução se tornam mais claros para nós.

```
if delta < 0 then
  printf('Raízes complexas!');
else
  r1 = (-b + sqrt(delta))/(2*a);
  r2 = (-b - sqrt(delta))/(2*a);
  printf('r1=%g e r2=%g.',r1,r2)
end
```

Mais legível

```
if delta < 0 then
printf('Raízes complexas!');
else
r1 = (-b + sqrt(delta))/(2*a);
r2 = (-b - sqrt(delta))/(2*a);
printf('r1=%g e r2=%g.',r1,r2)
end
```

Menos legível

Figura 272: Ifs equivalentes para o Scilab, mas com diferenças de legibilidade devido à indentação

Indentação é o termo empregado para esta prática de deslocamento de blocos internos na disposição gráfica de um programa. Seu emprego é visto como absolutamente essencial por toda a indústria de desenvolvimento de software. A Figura 272 e a Figura 273 mostram comparações entre comandos digitados com e sem o emprego de indentação.

```
if Nota >= 90 then
  Conceito = 'A';
else
  if Nota >= 80 then
    Conceito = 'B';
  else
    if Nota >= 70 then
      Conceito = 'C';
    else
      if Nota >= 60 then
        Conceito = 'D';
      else
        if Nota >= 40 then
          Conceito = 'E';
        else
          Conceito = 'F';
        end
      end
    end
  end
end
```

Mais legível

```
if Nota >= 90 then
Conceito = 'A';
else
if Nota >= 80 then
Conceito = 'B';
else
if Nota >= 70 then
Conceito = 'C';
else
if Nota >= 60 then
Conceito = 'D';
else
if Nota >= 40 then
Conceito = 'E';
else
Conceito = 'F';
end
end
end
end
end
```

Menos legível

Figura 273: Outro exemplo da influência da indentação na legibilidade de um comando

7.9 Geração e Impressão de uma Tabuada de Multiplicação

Vamos agora desenvolver um programa que produza uma tabuada de multiplicação, como mostrado na Figura 274.

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Figura 274: Tabuada de Multiplicação

Um problema como este se resolve com dois `for` aninhados: um externo, para as linhas, e um interno, para as colunas de cada linha, o que é feito pelo programa Tabuada1.sce, mostrado na Figura 275.

```
// Tabuada de multiplicação
for linha = 1:9
  for coluna = 1:9
    printf("%g",linha*coluna);
  end
end
```

Figura 275: O programa Tabuada1.sce

Ao executar este programa verificamos entretanto que sua saída está ininteligível:

```
12345678924681012141618369121518212...
```

formando uma longa sequência de algarismos sem separação, todos em uma única linha. O que houve? Esquecemos de orientar o Scilab para mudar de linha, e também para, dentro de cada linha, separar cada coluna.

```
// Tabuada de multiplicação
for linha = 1:9
  for coluna = 1:9
    printf("%3g",linha*coluna);
  end
  printf("\n");
end
```

Figura 276: O programa Tabuada2.sce

O programa Tabuada2.sce resolve estes problemas, com a inserção de um `printf("\n")`, executado ao término da impressão de cada linha, e com o código de formato `%3g` que fixa 3 colunas para a impressão de cada produto.

7.10 Arquivos

Arquivos são unidades de armazenamento de dados não-voláteis, que sistemas operacionais como Windows ou Linux permitem que sejam recuperados pelo nome e pela posição em uma organização hierárquica de diretórios. Um arquivo é criado por um programa, e pode ser lido e modificado por outros programas, programas que muitas vezes são executados em outros computadores. Arquivos são imprescindíveis para o tratamento de grandes volumes de dados.

Operações fundamentais com arquivos que podem ser realizadas por programas são:

- a leitura, que consiste na atribuição a variáveis de valores presentes no arquivo, e
- a escrita, que consiste na gravação no arquivo de valores de variáveis.

Existem muitos tipos de arquivos que podem ser manipulados por programas Scilab, mas neste curso iremos aprender somente a trabalhar com arquivos de texto, isto é, arquivos que normalmente são legíveis por humanos, e que podem ser editados usando programas como o Bloco de Notas do Windows.

Um comando muito útil para localização de arquivos no Scilab é o **`uigetfile`**, que permite ao usuário selecionar um arquivo navegando nos diretórios do sistema operacional. A Figura 277 mostra a janela de navegação para escolha de um arquivo aberta no Windows Vista após a execução de um comando **`uigetfile`**.

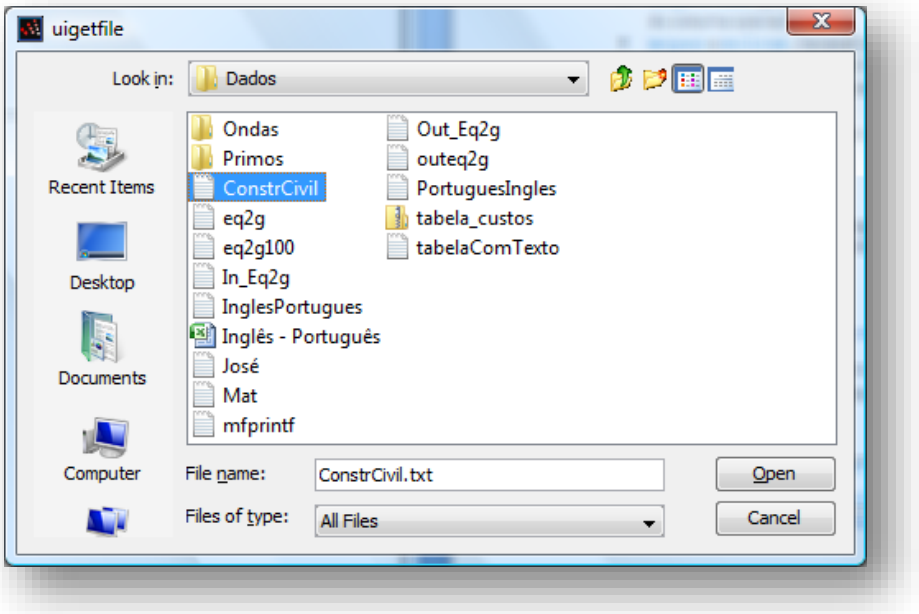


Figura 277: Janela de navegação e escolha de arquivos produzida com o comando uigetfile

O usuário pode escolher um entre os nomes de arquivos listados na janela, ou optar por um novo nome para um arquivo ainda não existente. O comando **`uigetfile`** retorna um string com a rota completa – isto é, desde o diretório raiz – do arquivo escolhido. Isso pode resultar em strings longos como o atribuído à variável **`f`** na Figura 278.

```
-->f = uigetfile(title="Escolha um arquivo:")
 f  =
 C:\Users\Osvaldo\Documents\dcc\Ensino\dcc001\Scilab\Dados\ConstrCivil.txt
```

Figura 278: String com rota completa obtido com `uigetfile`

Outras formas de uso do comando **`uigetfile`** são:

- **`Arq = uigetfile()`**
  - Mais simples para programar, mas nem tão bom para o usuário;
- **`Arq = uigetfile("*.txt",pwd(),"Escolha um arquivo")`**
  - Um pouco mais complicado, porém pode oferecer mais conforto para o usuário, pois:
  - Só são mostrados os arquivos selecionados por um filtro; no caso, o filtro é **`"*.txt"`**, que seleciona somente arquivos com terminação **`.txt`**, e

- a janela de escolha do arquivo tem o título `"Escolha um arquivo"`,
- e exibe inicialmente o diretório corrente do Scilab, o que pode evitar um aborrecido trabalho de navegação na estrutura de diretórios. A função `pwd()` retorna o diretório corrente do Scilab.

Outros comandos úteis são o `uigetdir`, que serve para a localização de diretórios, e o `listfiles`, que retorna um vetor de strings com os nomes de todos os arquivos em um diretório. A Figura 279 um exemplo de uso conjunto destes dois comandos.

```
-->d = uigetdir()
 d  =

 C:\Documents and Settings\osvaldo\My Doc
      uments\My Dropbox\Ensino\dcc001\201
      2-2\Módulo 05

-->arquivos =listfiles(d)
 arquivos  =

!Módulo5-2012-2.pptx     !
!                         !
!Módulo5-2012-2.pdf      !
!                         !
!Módulo 05.zip           !
!                         !
!Eq2g3.sce                !
!                         !
!Eq2g2.sce                !
!                         !
!Eq2g1.sce                !
!                         !
!Eq2g.sce                 !
```

Figura 279: Exemplo de uso dos comandos `uigetdir` e `listfiles`

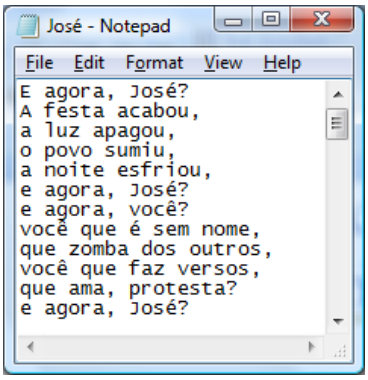
7.11 Matrizes de Strings e Arquivos

É possível ler um arquivo e transformar todas as suas linhas em um vetor coluna de strings, cada uma de suas linhas sendo lida em um elemento do vetor. Para isso deve ser usado o comando `mgetl(f)`, onde `f` é um nome de arquivo, normalmente obtido com `uigetfile`.

```
f = uigetfile();
linhas = mgetl(f);
```

Figura 280: Leitura de um arquivo como um vetor coluna de strings usando o comando `mgetl`

A Figura 280 mostra um uso típico do comando `mgetl`, precedido pela localização e da abertura do arquivo, e seguido pelo fechamento do arquivo.




```
-->linhas
 linhas  =
!E agora, José?            !
!A festa acabou,           !
!a luz apagou,             !
!o povo sumiu,             !
!a noite esfriou,          !
!e agora, José?            !
!e agora, você?            !
!você que é sem nome,      !
!que zomba dos outros,     !
!você que faz versos,      !
!que ama, protesta?        !
!e agora, José?            !
```

Figura 281: Arquivo fonte visto com o Bloco de Notas e vetor de strings obtido com o comando mgetl

Na Figura 281 nós vemos o efeito desses comandos quando o arquivo escolhido para leitura é o mostrado na parte esquerda da figura, com o famoso poema do Drummond.

7.12 Matrizes Numéricas e Arquivos

O Scilab oferece os comandos **fscanfMat** e **fprintfMat** que permitem a leitura e a gravação de matrizes em arquivos . Estes comandos só podem ser usados para arquivos que contêm ou conterão somente números em formato tabular, com exceção das primeiras linhas que podem conter textos. Os arquivos são lidos ou gravados com uma única execução desses comandos, que dispensam as operações de abertura e de fechamento de arquivos.

O comando **fprintfMat(arq, M, "%5.2f", Cabecalho)** grava a matriz numérica **M** no arquivo cujo nome é o valor (string) de **arq**. Esse arquivo em suas primeiras linhas irá conter os strings que são os elementos do vetor coluna de strings chamado **Cabecalho**. Cada elemento de **M** é gravado com o formato **"%5.2f"**. O vetor **Cabecalho** normalmente é usado para uma explicação sobre os campos presentes no arquivo.

```
a = [1 2 3; 4 5 6; 7 8 9];
arq = uigetfile();
Cabecalho = [" Meus Dados "; "Col1 Col2 Col3"]
fprintfMat(arq,a,"%5.2f",Cabecalho);
```

Figura 282: Gravação de uma matriz em um arquivo com fprintfMat

Na Figura 282 nós vemos um exemplo de uso do comando **fprintfMat**. O resultado deste programa é um arquivo como o mostrado na Figura 283.

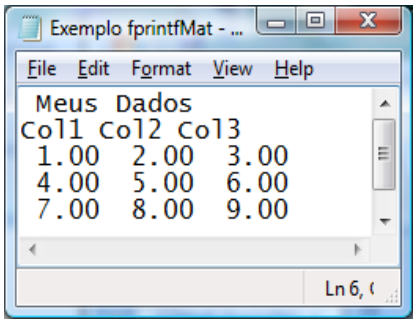


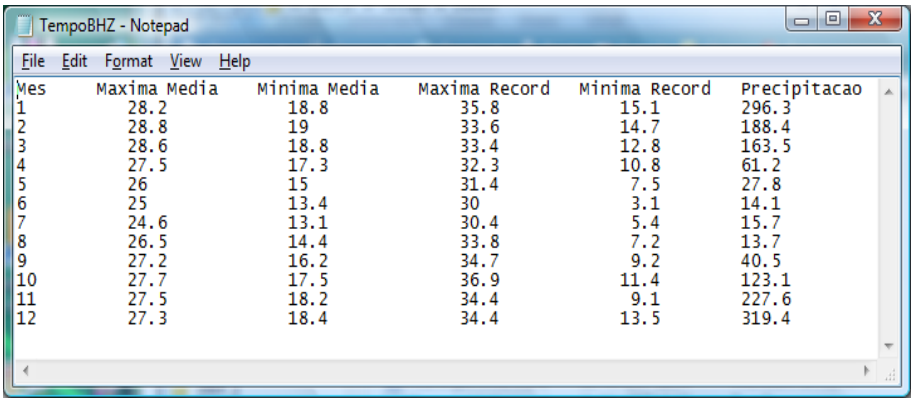
Figura 283: Arquivo gerado pelo programa da Figura 282

Se agora executarmos o programa da Figura 284, escolhendo como entrada o arquivo da Figura 283, iremos obter uma matriz **m** idêntica à matriz **a** gravada anteriormente.

```
arquivo = uigetfile();
m = fscanfMat(arquivo)
```

Figura 284: Leitura de uma matriz com o comand fscanfMat

Vamos agora ver um exemplo de aplicação de matrizes e arquivos. A Figura 285 mostra o arquivo TempoBHZ.txt exibido com o Bloco de Notas. Cada linha do arquivo contém o número de um mês, a temperatura média máxima do mês, a temperatura mínima média, a máxima record, a mínima record, e a precipitação do mês em milímetros de chuva.



Mes	Maxima Media	Minima Media	Maxima Record	Minima Record	Precipitacao
1	28.2	18.8	35.8	15.1	296.3
2	28.8	19	33.6	14.7	188.4
3	28.6	18.8	33.4	12.8	163.5
4	27.5	17.3	32.3	10.8	61.2
5	26	15	31.4	7.5	27.8
6	25	13.4	30	3.1	14.1
7	24.6	13.1	30.4	5.4	15.7
8	26.5	14.4	33.8	7.2	13.7
9	27.2	16.2	34.7	9.2	40.5
10	27.7	17.5	36.9	11.4	123.1
11	27.5	18.2	34.4	9.1	227.6
12	27.3	18.4	34.4	13.5	319.4

Figura 285: O arquivo TempoBHZ.txt

Nós queremos fazer um programa que:

- Leia os dados desse arquivo;
- Extraia desses dados vetores correspondentes a cada uma das colunas, conforme a Figura 285;
- Gere um gráfico que exiba simultaneamente as curvas de máxima média, mínima média, máxima record e mínima record.

Examinando o arquivo de entrada nós vemos que ele tem um formato adequado para leitura com **`fscanfMat`**, pois tem uma linha de cabeçalho, e os dados restantes são todos numéricos e dispostos em um formato tabular.

```
arqClima = uigetfile();
ClimaBH = fscanfMat(arqClima);
MaxMed = ClimaBH(:,2); // MaxMed = 2a coluna
MinMed = ClimaBH(:,3); // MinMed = 3a coluna
MaxRec = ClimaBH(:,4); // MaxRec = 4a coluna
MinRec = ClimaBH(:,5); // MinRec = 5a coluna
Precip = ClimaBH(:,6); // Precip = 6a coluna
plot2d([1:12],[MaxMed MinMed MaxRec MinRec],...
leg="MaxMed@MinMed@MaxRec@MinRec")
xtitle("Temperaturas Mensais em BH","Mês","Graus C");
```

Figura 286: O programa ClimaBHZ.sce

A Figura 286 mostra o programa ClimaBHZ.sce que atende a essa especificação. Como você pode ver, o programa é bastante simples, com a descoberta e leitura do arquivo, seguida das extrações das colunas, e seguida da geração do gráfico. É tão simples que aproveitamos para introduzir duas novas técnicas que podem melhorar a apresentação de um gráfico:

- o parâmetro extra de `plot2d`, `leg="MaxMed@MinMed@MaxRec@MinRec"`, que gera legendas para cada curva em no gráfico, e
- o comando `xtitle`, que determina títulos para o gráfico e para cada um dos eixos.

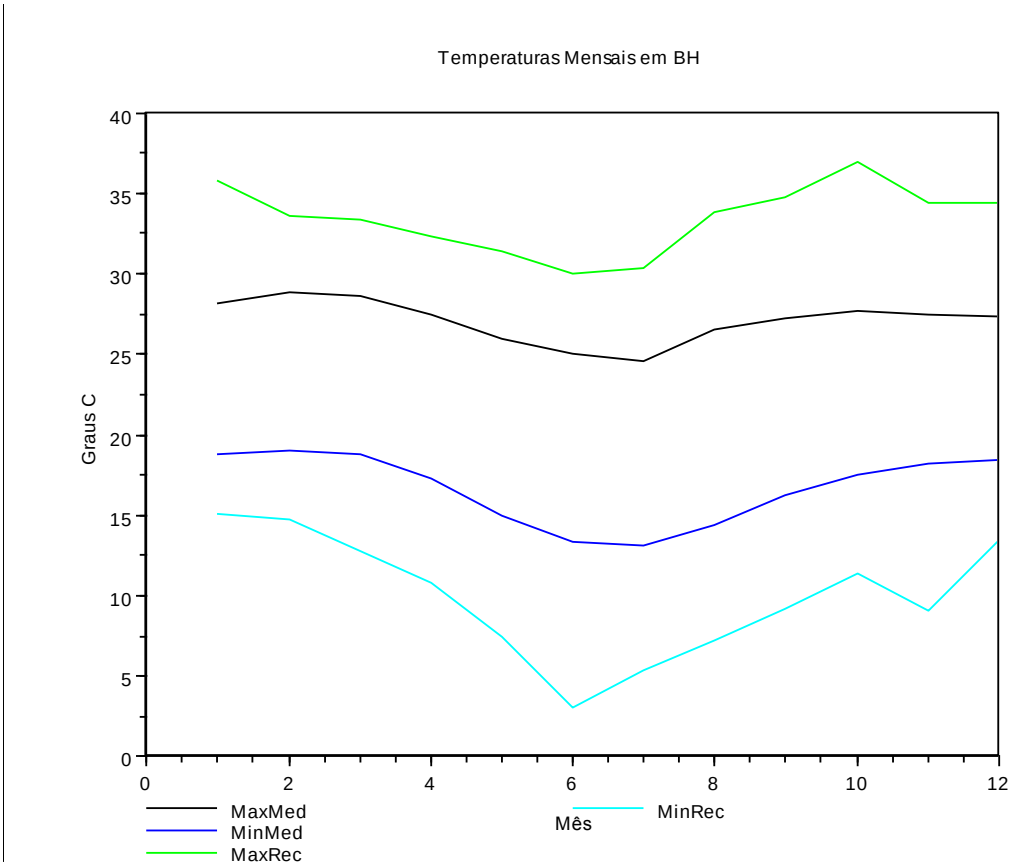


Figura 287: Gráfico gerado pelo programa da Figura 286

Na

Figura 287 você pode ver o efeito destas técnicas sobre o gráfico.

7.13 Desenhando Mapas

Para se desenhar um polígono usando o Scilab basta colocar as coordenadas de seus vértices em vetores `x` e `y` de mesmo tamanho, e executar `plot2d(x,y)`. Por exemplo, a sequência de comandos executados na console do Scilab

```
--> x = [1 2 3 4 2 1];
--> y = [1 2 1 2 4 1];
--> plot2d(x,y,rect=[0 0 6 6]);
```

(onde o parâmetro `rect` indica as coordenadas de um retângulo que determina os limites de exibição) produz o gráfico mostrado na Figura 288.

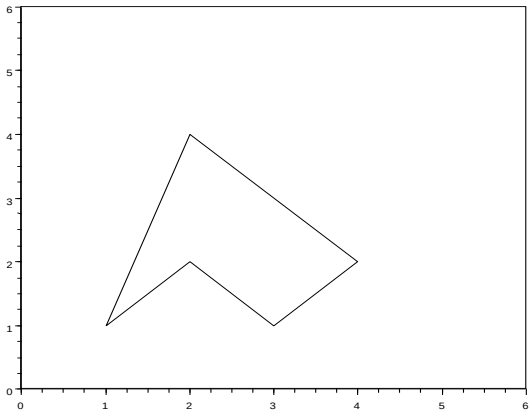


Figura 288: Um polígono

Se quisermos acrescentar outro polígono, podemos fazer

```
--> w = [4 5 5 4];
--> z = [0.5 0.5 1.5 0.5];
--> plot2d(w,z);
```

obtendo o gráfico mostrado na Figura 289.

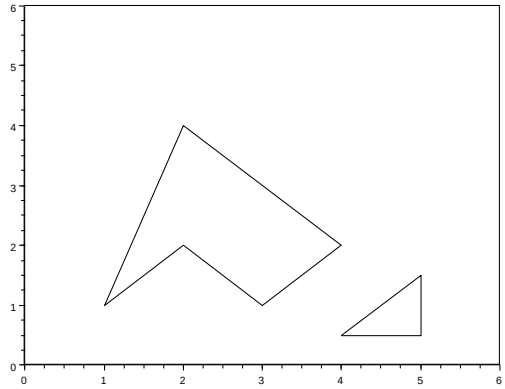


Figura 289: Dois polígonos

Repare que foram utilizados dois pares de vetores, um para cada polígono desenhado. Quando temos muitos polígonos a desenhar, como é o caso do mapa que faremos em seguida, torna-se interessante representar todos os polígonos em um único par de vetores com as coordenadas de todos os vértices, com a exibição feita por um único comando `plot2d`. É preciso entretanto encontrar um meio de informar ao Scilab quais são os pontos que separam dois polígonos. Isto porque se fizermos

```
--> X = [x w];
--> Y = [y z];
--> plot2d(X,Y,rect=[0 0 6 6]);
```

vamos obter o desenho da Figura 290, onde o ponto final do primeiro polígono e o ponto inicial do segundo polígono foram (naturalmente) “emendados” pelo Scilab.

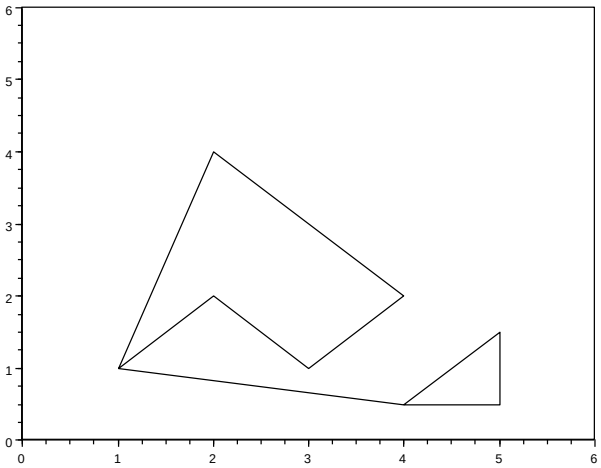


Figura 290: Efeito não desejado ao traçar dois polígonos

Este problema é resolvido inserindo-se pares de pontos com coordenadas iguais a **%nan** (representação de *not a number* em ponto flutuante; veja a Seção 2.1) para indicar ao Scilab os pontos de separação entre dois polígonos. Experimente agora fazer, na console do Scilab,

```
--> X = [x %nan w];
--> Y = [y %nan z];
--> clf
--> plot2d(X,Y,rect=[0 0 6 6]);
```

e você obterá a figura desejada, com polígonos separados. O comando **clf** limpa a janela gráfica do Scilab.

Um mapa simples pode ser representado por uma tabela de pares latitude-longitude. Em princípio estas tabelas podem ser utilizadas diretamente para o desenho dos mapas utilizando a função **plot2d**, mas, como já vimos, é necessário algum mecanismo para separar os polígonos isolados representando cada ilha, lago ou continente.

No nosso site você pode encontrar o arquivo **world.txt** (que obtivemos na referência [46], e de onde retiramos grande parte desta seção). Se você abrir este arquivo com um editor de arquivos texto como o PSPad[47] (o Bloco de Notas serviria, mas para trabalhar com arquivos texto, nós preferimos o PSPad porque ele possibilita um maior controle sobre formatos e codificações UTF-8, Windows, etc.), você verá que o valor 9999 foi utilizado como codificação para a separação de polígonos.

[46] "Applications with Scilab, Maxima, Geogebra", acessado 28 de setembro de 2011, http://www.wolffdata.se/.

[47] "Download PSPad - free unicode developer editor, handles near any syntax like HTML, PHP, XHTML, JavaScript, ASP, Perl, C and many other languages with HEX editor, multilanguage interface", acessado 6 de março de 2013, http://www.pspad.com/en/download.php.

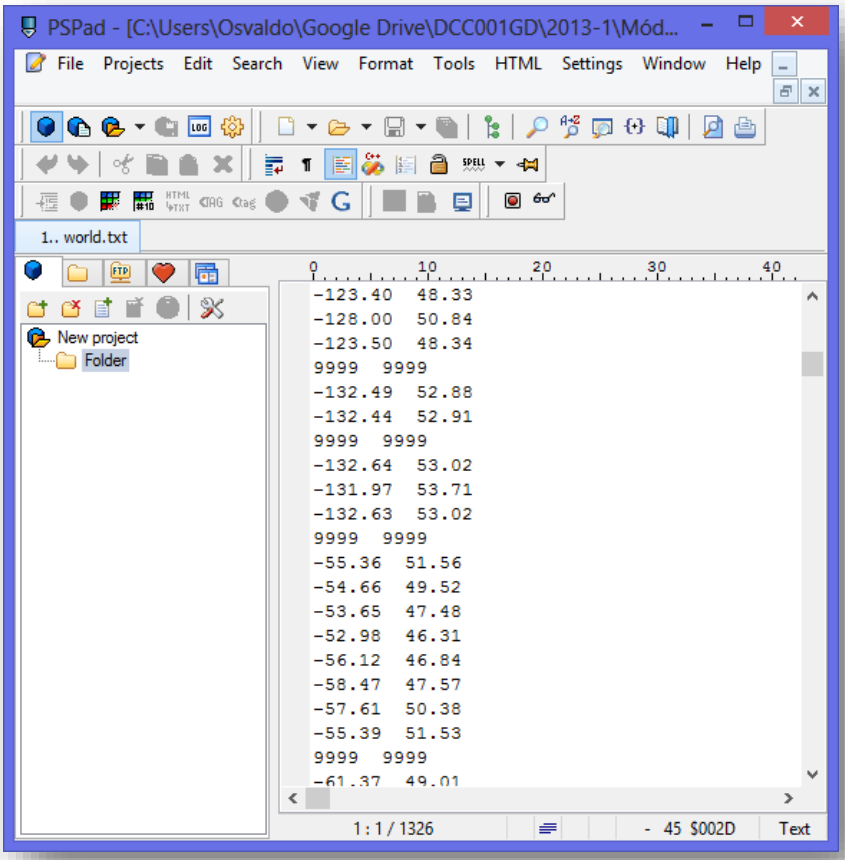


Figura 291: Arquivo world.txt aberto com o editor PSPad

e abra-o utilizando o Bloco de Notas (Notepad). Um pequeno exame desse arquivo basta para verificar que a separação entre os polígonos está ali representada pelas coordenadas com valores 9999. Isso se deve a uma restrição do comando **`fscanfMat`**, que só é capaz de ler números.

Faça agora um programa que

1) Leia este arquivo em uma matriz **`Mapas`**. Use para isto as funções **`uigetfile`** e **`fscanfMat`**.
2) Extraia desta matriz dois vetores, **`Lat`** e **`Long`**, correspondendo às duas colunas da matriz lida.
3) Substitua nos vetores **`Lat`** e **`Long`** os valores 9999 por **`%nan`**.
4) Use **`plot2d`** para obter o desenho do mapa representado no arquivo.

8 Funções

Funções são:

- uma ferramenta de modularização da mais alta importância para a programação, e
- um mecanismo linguístico para a prescrição de comportamentos repetitivos.

Vamos inicialmente mostrar o valor de funções como ferramenta de modularização, o que permite o reaproveitamento de código, a divisão de tarefas em projetos maiores de programação, e torna programas mais legíveis.

8.1 Funções e Reaproveitamento de Código

Para ilustrar o uso de funções, vamos desenvolver um programa que lê dois inteiros, n e k, e que calcula e imprime o número de combinações de n elementos tomados k a k, dado pela fórmula

$$\binom{n}{k} = \frac{n!}{(n-k)!\,k!}$$

Temos três fatoriais a calcular, e para isso, vamos procurar reaproveitar o código que conhecemos para o cálculo do fatorial, mostrado na Figura 292.

```
fat = 1;
for i = 1:n
  fat = fat*i;
end
```

Figura 292: Trecho de código para o cálculo do fatorial

A adaptação deste código aos três fatoriais necessários para o cálculo do número de combinações nos leva ao programa mostrado na Figura 293, onde cada cálculo de fatorial está destacado, e onde o comando de saída foi omitido.

```
n=input("n="); k=input("k=");
fat_n = 1; // Cálculo do fatorial de n
for i = 2:n
  fat_n = fat_n * i;
end
fat_n_k = 1; // Cálculo do fatorial de n-k
for i = 2:(n-k)
  fat_n_k = fat_n_k * i;
end
fat_k = 1; // Cálculo do fatorial de k
for i = 2:k
  fat_k = fat_k * i;
end
nComb = fat_n/(fat_n_k * fat_k)
```

Figura 293: Programa para cálculo do número de combinações de n k a k

Você deve reparar que foram feitas três adaptações do código, uma para cada fatorial a ser calculado. Nós vamos procurar mostrar que com o uso de funções este programa se torna muito mais claro. Para isso vamos dividir o programa em duas partes: o *programa principal* e a *função*.

```
function fat = fatorial(v)
  fat = 1;
  for i = 1:v
    fat = fat*i
  end
endfunction
```

Figura 294: Código da função `fatorial`

```
//Combinações de n k a k
exec("fatorial.sci");
n = input("n = ");
k = input("k = ");
fat_n = fatorial(n);
fat_k = fatorial(k);
fat_n_k = fatorial(n-k);
Comb_n_k = fat_n/(fat_k * fat_n_k);
printf("Comb(%d,%d) = %d",n,k,Comb_n_k)
```

Figura 295: Código do programa principal para cálculo do número de combinações de n k a k

O comando `exec` é um detalhe técnico: serve para indicar ao Scilab que a função `fatorial` será utilizada pelo programa. O programa da Figura 295 faz a mesma coisa que o da Figura 293, mas ao invés de conter três trechos de código para o cálculo dos fatoriais, contém três *chamadas* da função `fatorial`, e um único código para esta função, mostrado na Figura 296.

O código na Figura 295 recebe a designação de *programa principal*. A execução de um programa com funções se inicia pelo programa principal. A execução de uma chamada transfere o controle para a função; ao término da execução da função, o controle é devolvido para o ponto de chamada, em uma operação que chamamos de *retorno da função*.

O efeito das adaptações usadas para os diferentes fatoriais é obtido pelo uso de diferentes *parâmetros* da função a cada chamada, o que propicia um reaproveitamento de código muito mais elegante. A leitura do programa principal é também muito mais fácil, pois a intenção do programador se torna mais clara. Para alguém que não tivesse construído o programa da Figura 293, a percepção da similaridade entre os trechos de cálculo do fatorial pode não ser óbvia, e requer esforço de verificação. Para o próprio programador, as substituições utilizadas para a construção do programa da Figura 293 são uma fonte de enganos.

O cálculo de combinações é uma operação que pode ser aproveitada em outras ocasiões. Como vimos, o código de uma função é mais facilmente reaproveitado do que o código de um programa. Uma boa idéia é então transformar o programa da Figura 304 em uma função, o que resulta no código mostrado na Figura 296.

```
function nComb = Combinacoes(n,k)
  nComb = fatorial(n) /  (fatorial(n-k) * fatorial(k))
endfunction
```

Figura 296: Uma função para o cálculo do número de combinações de n k a k

Um programa equivalente ao da Figura 304 está mostrado na Figura 297. Você deve reparar no encadeamento de chamadas: o programa principal chama a função **Combinacoes**, que por sua vez contém três chamadas para a função **fatorial**.

```
exec("Combinacoes.sci")
exec("fatorial.sci")
n=input("n="); k=input("k=");
printf("nComb(%d,%d) = %d",n,k,Combinacoes(n,k))
```

Figura 297: Um programa principal que usa a função Combinacoes

A Figura 298 ilustra o encadeamento das chamadas neste programa.

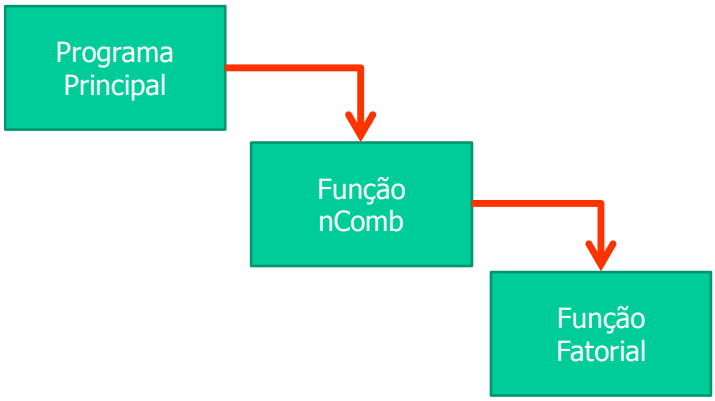


Figura 298: Encadeamento de chamadas

8.2 Forma e Funcionamento

```
function fat = fatorial(n)
  fat = 1;
  for i = 1:n
    fat = fat*i;
  end
endfunction
```

Figura 299: Palavras-chave na definição de uma função

Funções são definidas com o uso das palavras-chave **function** e **endfunction**, que delimitam o código da função.

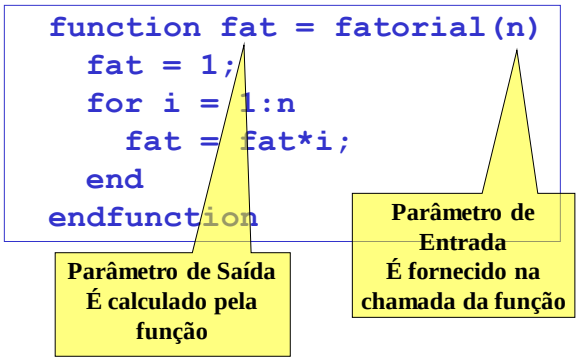


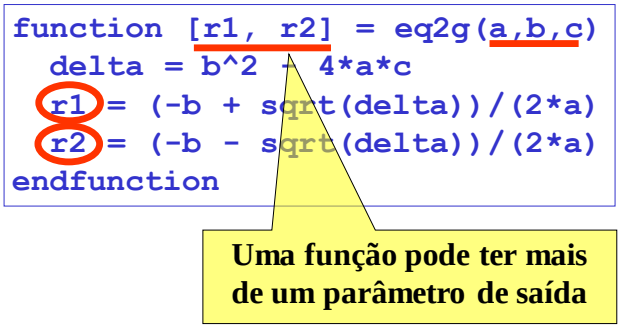
Figura 300: Parâmetros formais de entrada e de saída da função

A função **fatorial** possui um *parâmetro formal de entrada*, **n**, e um *parâmetro formal de saída*, **fat**. Parâmetros formais são definidos no *cabeçalho* da função, onde são utilizados como variáveis normais. Entretanto, parâmetros formais só ganham existência durante a

execução da função; valores assumidos por parâmetros formais em execuções anteriores não têm qualquer significado.

Por contraste, os parâmetros usados em chamadas de uma função são chamados de *parâmetros reais*; são expressões que produzem valores. No início da execução da função cada parâmetro formal recebe o valor do parâmetro real correspondente; a correspondência é estabelecida pela ordem dos parâmetros.

A execução da função deve calcular um valor para o parâmetro formal de saída; este valor é utilizado na expressão que contém a chamada da função. Uma função pode ter mais de um parâmetro formal de saída; a Figura 301 mostra um exemplo de definição e de chamada de uma função com dois parâmetros formais de saída.


function [r1, r2] = eq2g(a,b,c)
delta = b^2 - 4*a*c
r1 = (-b + sqrt(delta))/(2*a)
r2 = (-b - sqrt(delta))/(2*a)
endfunction
Uma função pode ter mais de um parâmetro de saída

Chamada da função eq2g

```
[raiz1,raiz2] = eq2g(x,y,z)
```

Figura 301: Função com dois parâmetros formais de saída

Uma função cria um espaço novo para variáveis, que podem ter nomes iguais aos de variáveis já definidas no programa principal. Variáveis criadas pela execução do corpo de uma função são chamadas *variáveis locais*. Na função da Figura 295 a variável `i` é uma variável local; o programador desta função não precisa se preocupar com qualquer outra variável de mesmo nome definida no programa principal ou em outra função. Essa delimitação de escopo de variáveis é uma propriedade essencial para permitir o desenvolvimento de funções por programadores independentes; se não existisse, todos os programadores participantes de um projeto de maior vulto teriam que se preocupar em escolher nomes de variáveis que não fossem utilizados por seus colegas.

Parâmetros de entrada de uma função podem ser valores numéricos, strings, valores lógicos, matrizes de qualquer tipo, e até mesmo uma outra função. A Figura 302 mostra um exemplo de uma função que recebe uma função `f` como parâmetro e que faz o seu gráfico para um vetor `x`.

```
function PlotaPontos(f,x)
  y = f(x);
  plot2d(x,y,style=-1);
endfunction
```

```
function y = MinhaFunc(x)
  y = exp(- x .^ 2);
endfunction
```

```
// Testador PlotaPontos
exec("PlotaPontos.sci");
exec("MinhaFunc.sci");
x = 0:0.1:2*%pi;
PlotaPontos(sin,x);
PlotaPontos(MinhaFunc,x);
```

Figura 302: Uma função que faz o gráfico de outra função recebida como parâmetro

8.3 Funções, arquivos fonte e o Scilab

Uma função é normalmente escrita em um arquivo com o mesmo nome da função, e com a extensão .sci, distinta dos arquivos com programas principais Scilab, que têm a extensão .sce.

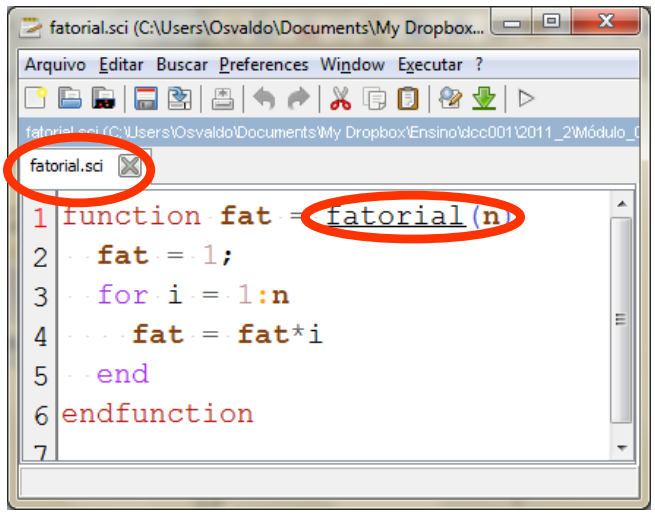


Figura 303: Um arquivo com uma função aberto no SciNotes

A Figura 303 mostra um arquivo com a função `fatorial` aberto no SciNotes, onde você pode reparar que o arquivo e a função têm o mesmo nome, sendo que o arquivo tem a extensão .sci.

A mera presença em seu computador de um arquivo com uma função não basta para que o Scilab saiba de sua existência. É necessário incorporar a função ao Scilab, o que pode ser feito com o comando

```
exec(<nome do arquivo com a função>)
```

que deve ser colocado no programa principal. Um exemplo de uso do comando **`exec`** está mostrado na Figura 304.

```
exec("fatorial.sci")
n=input("n="); k=input("k=");
nComb = fatorial(n) / (fatorial(n-k) * fatorial(k))
```

Figura 304: Programa principal para o cálculo de combinações com o comando exec

Atenção: o arquivo com a função deve estar no mesmo diretório em que se encontra o programa principal, e este deve ser o diretório corrente do Scilab (veja a Seção **Erro! Fonte de referência não encontrada.**).

8.4 Funções e Encapsulamento de Detalhes (e mais sobre arquivos)

Funções servem também para isolar detalhes de uma aplicação em uma parte bem definida de um programa. Vamos ilustrar isso mostrando mais possibilidades de tratamento de arquivos no Scilab, conhecimento que em si é também bastante útil.

Nós já vimos na Seção 7.11 como ler e escrever todas as linhas de um arquivo com os comandos `mgetl` e `mputl`, e na Seção 7.12 como ler e escrever arquivos para e de matrizes numéricas, com os comandos `fscanfMat` e `fprintfMat`. Isso não é o suficiente para o tratamento de arquivos com formatos de uso muito frequente como os que veremos aqui: arquivos com campos em colunas pré-fixadas, e arquivos com campos separados por vírgulas ou por outros caracteres. Para isso vamos precisar de outros comandos, de uso um pouco mais complicado mas que permitem tratar arquivos de forma mais flexível.

O comando `d_arq = mopen(nomeArq,"r")` *abre* um arquivo para leitura; se o segundo parâmetro de `mopen` for `"w"`, o arquivo será aberto para escrita. Aqui a variável `nomeArq` deve conter um string com o nome completo de um arquivo, provavelmente obtido com `uigetfile`. A variável `d_arq` recebe como valor um descritor de arquivo, que deve ser usado em todas as operações de leitura ou de escrita no arquivo, e também para o seu fechamento. O comando `mclose(d_arq)` *fecha* um arquivo. Operações de leitura ou de escrita em arquivos devem ser feitas com arquivos abertos, e um programa correto fecha todos os arquivos que abriu (a forma `mclose("all")` pode ser útil para isso).

Tipicamente o tratamento de arquivos é feito como mostrado na Figura 305.

```
d = mopen(arquivo);
  // operações de leitura ou de escrita
mclose(d);
```

Figura 305: Uso típico de abertura e fechamento de arquivos

Abertura e fechamento de arquivos são operações necessárias pelo seguinte motivo. Quando não está em uso, um arquivo tem toda a sua informação gravada em disco ou em outra mídia não volátil. Para utilizar um arquivo, o sistema operacional deve trazer para a memória principal informações sobre o arquivo, como seu tamanho, as posições que fisicamente ocupa no disco (ou em outra mídia), e várias outras. Isso corresponde à abertura do arquivo. Nem todas as alterações sobre a informação do arquivo mantida na memória são imediatamente refletidas no disco; o fechamento do arquivo garante essa atualização.

```
201092916908 ROMULO CAVALCANTI REIS
201212078661 LAIS LUZ LARANJEIRA HENRIQUE
200219665183 DEBORAH CARSALADE MAGALHAES
201157086079 GUSTAVO SILVA PENA LELIS
201262618478 PEDRO PEREIRA MAURELLI
200929106821 BERNARDO VIEGAS SANTOS CARSALADE
```

Figura 306: Primeiras linhas de um arquivo com formato de colunas fixas

A Figura 306 mostra um exemplo típico de arquivo com formato de colunas fixas, descrito na Figura 307.

Colunas		Conteúdo
Inicial	Final	
1	4	Ano de registro
5	12	Número de cadastro
13	13	em branco
14	50	Nome

Figura 307: Descrição do arquivo da Figura 306

A Figura 308 mostra uma função para ler um arquivo com este formato, colocando seu conteúdo em vetores `Ano`, `Inscr` e `Nome`.

```
function [Ano,Inscr,Nome] = LeAnoInscrNomes(arq)
    k = 0;
    d_arq = mopen(arq,"r");
    while ~meof(d_arq)
        linha = mgetl(d_arq,1);
        k = k + 1;
        Ano(k) = eval(part(linha,1:4));
        Inscr(k) = part(linha,5:12);
        Nome(k) = stripblanks(part(linha,14:50));
    end
    mclose(d_arq);
endfunction
```

Figura 308: Função para leitura do arquivo da Figura 306

Temos aqui diversas observações e novidades:

- a variável `k` é usada como um contador de linhas; tem inicialmente o valor 0, e é incrementada a cada linha lida.
- a função `meof(<descritor de arquivo>)` retorna um booleano que indica se o fim do arquivo já foi alcançado (`eof` vem de *end of file*; o `m` é inexplicável);
- a função `mgetl` nós já conhecemos, mas aqui temos uma nova forma de uso. No comando `linha = mgetl(d_arq,1)` uma linha do arquivo de descritor `d_arq` é atribuída à variável `linha` (o comando `linha = mgetl(d_arq, 3)` colocaria em linha 3 linhas do arquivo);
- para extrair as informações contidas em cada linha, a função `part` é de extrema valia, pois com ela podemos obter strings que correspondem a uma parte de um outro string;
- a função `eval` é utilizada para obter o valor numérico de um string com algarismos, pontos e sinais decimais;
- a função `stripblanks` serve para retirar brancos das extremidades de um string:

```
-->length("    abc    ")
 ans  =
    11.

-->length(stripblanks("    abc    "))
 ans  =
    3.
```

A Figura 309 mostra uma função que faz a operação inversa, isto é, grava um arquivo com o formato da Figura 306 a partir de dados presentes em três vetores.

```
function GravaAnoInscrNomes(Ano,Inscricoes,Nomes,arq)
    n = size(Ano,"*");
    d_arq = mopen(arq,"w");
    for i = 1:n
        mfprintf(d_arq,"%4d%8s%40s\n", ...
                 Ano(i),Inscricoes(i),Nomes(i))
    end
    mclose(d_arq);
endfunction
```

Figura 309: Função para gravar um arquivo em formato de colunas fixas

Aqui nós utilizamos a função `mfprintf`, que é a versão de `printf` para gravar em um arquivos, usando seu descritor.

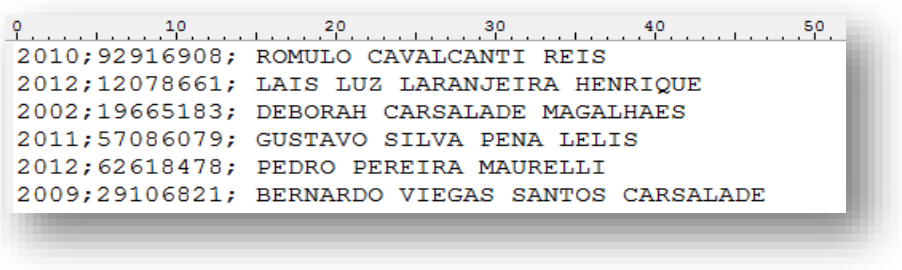


Figura 310: Primeiras linhas de um arquivo em formato CSV

Na Figura 310 nós vemos as mesmas informações do arquivo da Figura 310 em um outro formato muito utilizado, onde os campos encontram-se separados pelo caractere “;”. Este formato é conhecido como formato CSV, de *Comma Separated Values*, mesmo quando o separador não é uma vírgula.

```
function[Ano,Inscr,Nome] = LeAnoInscrNomesCSV(arq)
    d_arq = mopen(arq,"r"); k = 0;
    while ~meof(d_arq)
        linha = mgetl(d_arq,1);
        k = k + 1;
        tks = tokens(linha,[";"]);
        Ano(k) = eval(tks(1));
        Inscr(k) = tks(2);
        Nome(k) = stripblanks(tks(3));
    end
    mclose(d_arq);
endfunction
```

Figura 311: Função para leitura de um arquivo com o formato CSV da Figura 310

A Figura 311 mostra uma função que lê um arquivo com o formato da Figura 311. Aqui também o arquivo é lido linha por linha, com a função `tokens` sendo utilizada para obter os campos de uma linha como elementos de um vetor de strings.

```
function GravaAnoInscrNomesCSV(Ano,Inscricoes,Nomes,arq)
    n = size(Ano,"*");
    d_arq = mopen(arq,"w");
    for i = 1:n
        mfprintf(d_arq,"%4d;%8s; %-40s\n",...
                 Ano(i),Inscricoes(i),Nomes(i))
    end
    mclose(d_arq);
endfunction
```

Figura 312: Função para gravação de um arquivo em formato CSV

A gravação de um arquivo com o formato CSV pode ser feita facilmente com o comando `mfprintf`, como mostra a Figura 312.

8.5 Funções e Desenvolvimento Top-down

Uma técnica comum de programação é a utilização da chamada de uma função antes do desenvolvimento da própria função, técnica conhecida como desenvolvimento *top-down*. Em alguns momentos do desenvolvimento, o programador é capaz de especificar *o que* ele quer que a função faça, deixando para depois o trabalho de determinar *como* fazê-lo.

Vamos aqui ilustrar o emprego de desenvolvimento top-down para construir um programa que:

- Leia uma série de números inteiros maiores ou iguais a 2
- Para cada número lido, encontre o menor número primo que seja maior ou igual a ele. Por exemplo, se o número lido for 4, o programa deve encontrar o número primo 5; se for 11, o programa deve encontrar 11, que já é primo. Da matemática nós sabemos que o conjunto de números primos é infinito, ou seja, que sempre existe um número primo maior ou igual a um número dado.
- O programa deve terminar quando o usuário entrar com um número menor que 2.

Construir um programa que leia uma série de dados, faça alguma coisa com eles, e termine conforme o desejo do usuário, é uma idéia que já foi explorada por diversas vezes neste curso. Encontrar o menor primo maior ou igual a um número lido nos parece complicado, mas vamos dar início ao desenvolvimento do programa deixando isso para depois. Veja uma proposta na Figura 313.

```
n = input("n = (use n < 2 se quiser parar):");
while n >= 2
  // Encontra o menor primo >= n
  // e imprime o resultado
  printf("O menor primo >= %d é %d",...
  n,MenorPrimoMaiorOuIgualA(n))
  // Lê n
  n = input("n = (use n < 2 se quiser parar):");
end
```

Figura 313: Programa principal para Menor Primo >= n

Podemos ver que o programa principal cuida da interação com o usuário, empurrando o problema de se encontrar o menor primo para uma função `MenorPrimoMaiorOuIgualA`, que ainda não existe. Não existe, mas já demos a ela um nome significativo, e especificamos que:

- A função tem um único parâmetro de entrada, que é o número digitado pelo usuário
- A função deve retornar o número primo que desejamos.

Ou seja, já definimos o cabeçalho da função **MenorPrimoMaiorOuIgualA**. Muito bem, vamos agora encarar o seu desenvolvimento. O algoritmo é simples: vamos testar sequencialmente os inteiros a partir do número lido, parando ao encontrar um número primo. Sim, mas como vamos saber se um inteiro é primo? Não vamos nos preocupar com isso agora. Outra vez, vamos especificar uma função e adiar o problema. Veja a Figura 314.

```
function p = MenorPrimoMaiorOuIgualA(n)
  p = n;
  while ~Primo(p)
    p = p+1
  end
endfunction
```

Figura 314: A função MenorPrimoMaiorOuIgualA

Como funciona a função **Primo**, não sabemos ainda, mas especificamos que:

- a função tem um parâmetro formal de entrada, **n**, que é o número que queremos testar se é ou não primo;
- o resultado do teste é o parâmetro formal de saída, que deve ser %t se o número for primo, e %f senão.

Para saber se um número n é primo vamos decompô-lo como um produto de fatores inteiros satisfazendo $n = pq$. Um número inteiro n maior ou igual a 2 é primo se seus únicos fatores são 1 e o próprio n. A função **Primo** pode então ser escrita como mostrado na Figura 315,

```
function ehPrimo = Primo(n)
  ehPrimo = (n == MenorFator(n));
endfunction
```

Figura 315: A função Primo

Precisamos agora desenvolver a função **MenorFator**. O algoritmo que vamos utilizar é bastante direto: vamos usar um inteiro **p**, com valor inicial 2, e que iremos a cada passo de um loop incrementar e testar se ele é um divisor. Para este teste vamos usar a função Scilab **modulo(n,p)** , que calcula o resto da divisão de **n** por **p**. O loop para ao encontrar o primeiro divisor, o que sempre acontece pois **n** é divisível por **n**. Ao término do loop, **p** contém o menor divisor de **n**.

```
function p = MenorFator(n)
  p = 2;
  while modulo(n,p) <> 0
    p = p + 1;
  end
endfunction
```

Figura 316: A função MenorFator

Com isso nós terminamos o desenvolvimento do nosso programa. É importante que você observe os ganhos obtidos com esse enfoque:

- Ao desenvolver o programa principal, nós nos preocupamos com a interação com o usuário e, exceto pela definição de funcionalidade, pudemos esquecer do problema de encontrar o número primo que desejamos;

- Ao desenvolver a função **MenorPrimoMaiorOuIgualA**, nós esquecemos da interação com o usuário e nos preocupamos somente com a procura seqüencial por um primo a partir de um número dado; saber se um número é primo é também um problema deixado para outro momento;
- Ao desenvolver a função **Primo**, nosso foco é simplesmente descobrir um algoritmo para saber se um número inteiro é primo ou não; todo o contexto restante pode ser esquecido. Para isso o parâmetro de entrada **n** é comparado com o seu menor fator, e a descoberta deste menor fator é deixada para depois;
- Ao desenvolver a função **MenorFator**, a única preocupação é encontrar o menor divisor de um número inteiro.

É esta divisão de tarefas que permite o domínio de programas mais complexos. O desenvolvimento de cada função pode ser feito por um desenvolvedor em momentos diferentes, ou por pessoas distintas em uma mesma equipe.

8.6 Funções Recursivas

Vamos agora examinar o uso de funções para a prescrição de comportamentos repetitivos. Nós vimos que uma função pode chamar outra função, que pode também chamar outra função, que pode ainda chamar outra função, em um encadeamento de chamadas de profundidade arbitrária. Uma função pode também chamar a si própria, o que a torna uma *função recursiva*. Nós veremos ao longo desse curso que uma formulação recursiva é muitas vezes a forma mais natural para a descrição de algoritmos.

Como um primeiro exemplo, vamos mostrar uma função recursiva para o cálculo do fatorial de um número. Nós sabemos que

$$1! = 1$$

e que, para $n > 1$,

$$n! = n.(n-1)!$$

A função **fatorialR** na Figura 317 calcula o fatorial de um número usando de forma muito natural essas equações.

```
function fat = fatorialR(n)
  if n > 0 then
    fat = n*fatorialR(n-1)
  else
    fat = 1
  end
endfunction
```

Figura 317: Função recursiva para o cálculo do fatorial

Para compreender melhor o funcionamento da execução de uma função recursiva, considere o programa **FatorialR_Teste** da Figura 318.

```
// Teste de FatorialR
exec("FatorialR.sci");
n = input("n = ");
while n > 0 do
  printf("\n%d! = %d",n,FatorialR(n));
  n = input("n = ");
end
```

Figura 318: Programa TestaFatorialR.sce

Considere também a versão de `fatorialR` da Figura 319, onde também colocamos comandos `printf` envolvendo a chamada (recursiva) da função.

```
function fat = FatorialR(n)
  // Comente os printf para não imprimir
  printf("\nIniciando FatorialR(%d)",n);
  if n > 1 then
    fat = n * FatorialR(n-1);
  else
    fat = 1;
  end
  printf("\nRetornando Fatorial(%d) = %d",n,fat)
endfunction
```

Figura 319: A função fatorialR, instrumentada para acompanhamento de sua execução

A execução deste programa pode gerar a saída mostrada na Figura 320, onde acrescentamos chaves para indicar a correspondência entre chamadas e retornos da função.

```
n = 5

Iniciando FatorialR(5)
Iniciando FatorialR(4)
Iniciando FatorialR(3)
Iniciando FatorialR(2)
Iniciando FatorialR(1)
Retornando Fatorial(1) = 1
Retornando Fatorial(2) = 2
Retornando Fatorial(3) = 6
Retornando Fatorial(4) = 24
Retornando Fatorial(5) = 120
5! = 120
```

Figura 320: Saída do programa TestaFatorialR.sce

Chamadas e retornos de funções seguem um mecanismo clássico em computação, chamado de *pilha*. Em uma pilha de livros normalmente coloca-se um novo livro encima da pilha, e retira-se o livro no topo da pilha.

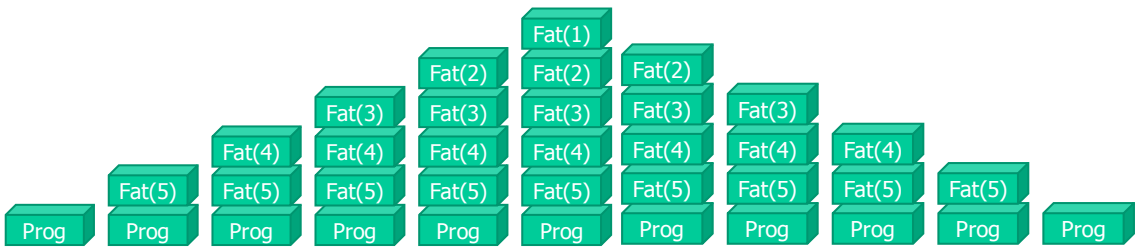


Figura 321: Pilha de execução de FatorialR

A Figura 321 ilustra a evolução da pilha de execução do programa TestaFatorialR.sce quando executado com `n == 5`. No início somente o programa principal está em execução. A primeira chamada de `FatorialR` é feita com o parâmetro real 5. Temos neste momento o programa principal e uma instância da função em execução. Mas o cálculo do fatorial de 5 exige outra chamada de `FatorialR`, desta vez com o parâmetro real 4, o que nos leva a ter em execução o programa principal e duas instâncias de `FatorialR`. Para o cálculo do fatorial de 4, precisamos do fatorial de 3, e assim vamos até chegarmos a uma pilha com o programa principal e cinco instâncias de `FatorialR`, que na última vez terá sido chamada com o parâmetro real 1.

Com o parâmetro real igual a 1 `FatorialR` retorna sem outra chamada recursiva, o que reduz a quatro o número de instâncias em execução. Isso marca o início de uma sequência de retornos e de desempilhamentos, até termos novamente somente o programa principal em execução.

A recursividade é portanto uma outra forma de se prescrever um comportamento repetitivo para um programa. É possível por exemplo formular o algoritmo de descoberta do menor valor presente em um vetor como uma função recursiva. Uma possibilidade usa a seguinte relação de recorrência:

- se `length(A) == 1`, o menor valor presente em `A` é `A(1)`
- se `length(A) > 1`, o menor valor presente em `A` é o menor entre `A(1)` e o menor valor presente em `A(2:length(A))`

o que nos leva a uma formulação recursiva do mesmo algoritmo que usamos na função Minimo. Outra relação de recorrência que pode ser usada é:

- se `length(A) == 1`, o menor valor presente em `A` é `A(1)`
- se `length(A) > 1`, o menor valor presente em `A` é o menor dentre (o menor valor presente na metade esquerda de `A`) e (o menor valor presente na metade direita de `A`)

A partir desta segunda relação de recorrência nós podemos derivar a função recursiva da Figura 322.

```
function m = MinimoR(x)
  if length(x) == 1 then
    m = x(1)
  else
    half = int(length(x)/2);
    minLeft = MinimoR(x(1:half));
    minRight = MinimoR(x(half+1:length(x)));
    if minLeft <= minRight then
      m = minLeft
    else
      m = minRight
    end
  end
endfunction
```

Figura 322: Função MinimoR, recursiva

9 Algoritmos, Pesquisa

9.1 Definição e Características

Um problema de transformação de informação descreve uma entrada de informação e propriedades requeridas de uma informação de saída. Um *algoritmo* resolve um problema de transformação de informação, sendo constituído por uma prescrição de passos que transformam uma informação de entrada em outra informação de saída. Cada passo prescrito deve ser uma operação garantidamente realizável, seja por operações elementares, seja por outro algoritmo. Um programa é a concretização de um algoritmo em uma linguagem executável. Nós lidamos com algoritmos desde o início deste texto, mas agora vamos dar um tratamento mais formal a este conceito.

Três questões são básicas para a caracterização de problemas de transformação de informação e algoritmos:

- *Especificação*: qual é exatamente o problema de transformação de informação que queremos resolver?
- *Correção*: um dado algoritmo resolve mesmo o problema proposto em sua especificação?
- *Eficiência*: com qual consumo de recursos computacionais (essencialmente, tempo e memória) o algoritmo executa a sua função?

Nós iremos a seguir detalhar um pouco mais cada uma dessas questões.

Especificações surgem de um processo de análise de uma necessidade de transformação de informação. Uma especificação não é estática; muitas vezes uma especificação é modificada durante e mesmo após o processo de desenvolvimento de um programa. Uma condição não prevista é muitas vezes a causa de uma alteração em uma especificação. Como um exemplo, no programa que vimos para a solução de muitas equações de 2º grau com coeficientes em um arquivo (**Erro! Fonte de referência não encontrada.**), nós não demos um tratamento especial para o caso de termos o primeiro coeficiente nulo, o que provoca uma divisão por zero no cálculo das raizes. Se em uma linha do arquivo de coeficientes tivermos $a = 0$, a divisão por zero provocará uma interrupção da execução do programa, sem o processamento das equações restantes. Se isso não for um comportamento aceitável, é necessário modificar a especificação para dar um tratamento adequado a esta situação, que poderia ser, por exemplo, ignorar a linha com $a = 0$, ou senão resolver a equação de primeiro grau resultante.

Em problemas reais é comum que a fase de especificação seja a etapa mais demorada e a mais cara de um projeto de desenvolvimento de um programa. Não existe situação pior para uma equipe de desenvolvimento do que, ao dar por terminado um sistema, constatar que ele não atende às necessidades do cliente, necessidades que, por deficiências no processo de análise, não foram explicitadas na especificação.

Apesar dessa importância, neste curso nós procuraremos lidar com problemas cuja especificação é bem simples, pois nosso objetivo aqui é a criação de uma cultura algorítimica para o aluno. Técnicas de análise e de especificação de sistemas são matérias que você poderá depois estudar em cursos mais avançados de programação ou de engenharia de software.

É possível verificar se um algoritmo atende a uma especificação por um exame de sua estrutura, com a construção de uma prova formal de sua correção. Na prática uma prova formal de correção só é viável para algoritmos muito pequenos. O que se faz é produzir uma argumentação informal da correção de um algoritmo.

Além desta argumentação, *testes* são usados para se ganhar convicção do bom funcionamento de um algoritmo concretizado em um programa. É entretanto importante ter em mente que testes podem descobrir erros, mas raramente podem garantir a sua ausência. Este problema vem do fato de que mesmo algoritmos muito simples têm tantas entradas possíveis que testes só podem cobrir uma ínfima fração dessas possibilidades.

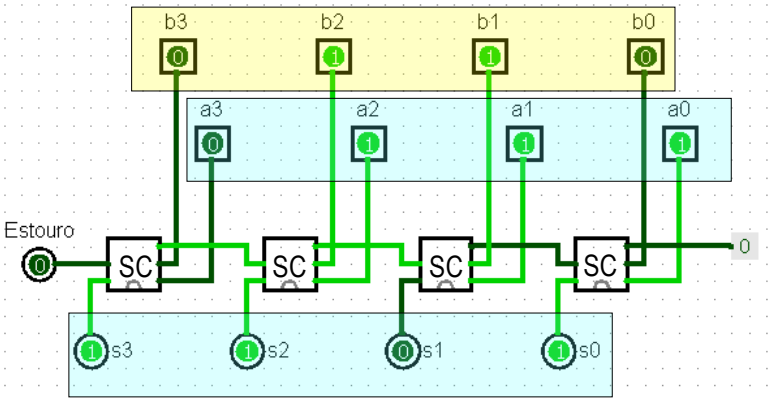


Figura 323: Um somador de 4 bits

Usando um exemplo retirado de (Dijkstra, 1972) vamos considerar uma transformação básica de informação, feita por um somador em cascata similar ao mostrado na Figura 323, mas com largura de 32 bits. Se alguém quiser considerar o somador como uma caixa preta a ser testada, teríamos 2^{64} (considerando as duas entradas de 32 bits cada) valores para as entradas a serem testadas, o que é claramente inexeqüível. Uma argumentação baseada na *estrutura* do somador é o que nos provê a convicção de sua correção. Se estamos convencidos da correção de um circuito de soma completa, como o mostrado na Figura 92 (e estamos, pela forma de desenvolvimento a partir da tabela da verdade), e se compreendemos a lógica do arranjo em cascata, não temos dificuldades para acreditar no bom funcionamento do somador de 32 bits.

A compreensão completa da estrutura de um programa grande – estamos falando de milhares ou mesmo milhões de linhas de código, produzidas por vários desenvolvedores – é por sua vez dificilmente atingível. Não é possível testar completamente um programa grande, e nem é possível compreender completamente sua estrutura. E agora, José? Bom, é isso mesmo. Não existe uma boa saída. A indústria de software investe em qualidade de desenvolvimento, em testes, mas quase sempre vende programas sem garantia.

Neste curso nós iremos trabalhar com programas pequenos, todos com menos de 100 linhas. A convicção de correção será fornecida tanto com argumentos estruturais como por testes.

Para atender a uma mesma especificação é possível encontrar algoritmos que apresentam enormes diferenças em sua eficiência. O temo *complexidade computacional*, ou simplesmente *complexidade*, é usado para designar como o uso de recursos computacionais por um algoritmo varia em função de seus dados de entrada. *Complexidade temporal* refere-se à eficiência em tempo de execução; *complexidade espacial* refere-se à eficiência no uso de memórias.

Na Ciência da Computação, um algoritmo tem complexidade maior que outro quando é menos eficiente, e não, como se poderia pensar com o uso habitual da palavra, mais complicado do que o outro. Na verdade, espera-se que algoritmos mais complicados tenham complexidade computacional menor do que a de algoritmos mais simples. Algoritmos com complexidade maior (menos eficientes) e mais complicados não têm nenhuma vantagem de uso, e inevitavelmente caem no esquecimento.

9.2 Fatoração

Para ganharmos algum sentimento de como algoritmos que satisfazem uma mesma especificação podem diferir em eficiência, nós vamos fazer alguns experimentos de medidas de tempo gasto com a fatoração de números inteiros, um problema de importância fundamental para a área de segurança da informação. A segurança da criptografia da maior parte dos sistemas atuais depende da dificuldade da fatoração de números semi-primos, isto é, números que são formados pelo produto de dois primos. Estamos aqui falando de números grandes: chaves criptográficas atuais têm costumeiramente 1024 ou 2048 bits, que correspondem a números com 308 ou 616 algarismos decimais.

O Scilab oferece a função `timer()` que permite medir o tempo gasto na execução de trechos de um programa. A primeira execução da função `timer()` zera e dispara um cronômetro; cada chamada subseqüente da função `timer()` retorna o valor em segundos do tempo decorrido desde a chamada anterior da função.

```
// Programa para fatorar números inteiros
exec("MenorFator.sci")
n = input("n = ");
while n >= 2
  timer(); p = MenorFator(n) ; tempoGasto = timer();
  // Imprime o resultado
  printf("\nTempo = %8.6fs, %6d é divisível por %6d",
tempoGasto,n,p);
  if n == p then
    printf(" **PRIMO**")
  end
  n = input("n = ");
end
```

Figura 324: O programa Fatora.sce

Na Figura 324 nós vemos o programa `Fatora.sce`, que lê números e encontra o menor fator divisor dos números lidos, destacando em sua saída os números primos. A verificação é feita pela função `MenorFator`, mostrada na Figura 316. Repare como a função `timer()` é utilizada no programa `Fatora.sce` para medir o tempo gasto na execução da função `MenorFator`.

Nosso primeiro experimento é muito simples. Vamos utilizar o programa `Fatora.sce` para encontrar o menor divisor de 131101, que é primo, e também de 131103, que não é.

```
n = 131101
Tempo = 3.062500s, 131101 é divisível por 131101 **PRIMO**

n = 131103
Tempo = 0.000000s, 131103 é divisível por      3
```

Figura 325: Tempos para fatorar números primos e não primos

Na Figura 325 nós vemos o resultado do experimento. Encontrar o menor fator de um número primo tomou mais de 3 segundos, enquanto o tempo tomado para encontrar o menor fator de um número divisível por 3 foi tão pequeno que a função `timer` retornou zero. Isso nos leva à importante observação de que o tempo gasto pode variar muito com a instância de um problema. É fácil entender o que aconteceu olhando o código da função `MenorFator`. Números primos são o pior caso para a função `MenorFator`. Na primeira chamada, foram feitas 131103 execuções da função `modulo`, e, na segunda, apenas 3. Com isso, uma primeira

observação: *algoritmos podem ter seu tempo de execução dependente da instância específica do problema que resolve*.

Muitas vezes nós estamos interessados na análise do pior caso de um algoritmo. Nós iremos em seguida relatar outros experimentos feitos com o programa Fatora.sce. Nestes experimentos nós vamos precisar de números primos, e para isso o arquivo 200000primos.txt, que contém os 200.000 primeiros números primos, é de grande utilidade. Diversos sites na Internet, como [48], contêm arquivos com números primos ou programas que geram números primos. A Figura 326 mostra as primeiras e últimas linhas deste arquivo.

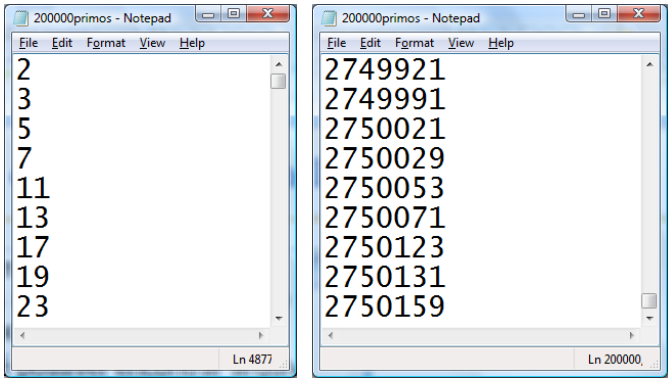


Figura 326: Primeiros e últimos números primos no arquivo 200000primos.txt

No segundo experimento, fizemos o programa `Fatora.sce` fatorar por diversas vezes se o número 131101, que é primo. Veja os resultados na Figura 327.

```
n = 131101
Tempo = 2.984375s, 131101 é divisível por 131101 **PRIMO**

n = 131101
Tempo = 3.078125s, 131101 é divisível por 131101 **PRIMO**

n = 131101
Tempo = 3.015625s, 131101 é divisível por 131101 **PRIMO**
```

Figura 327: Variações no tempo de execução de um programa com os mesmos dados de entrada

Você pode ver que um mesmo programa apresenta variações no tempo gasto para a fatoração do mesmo número primo. Isso se deve a uma série de fatores, o principal deles sendo o fato de que, em um sistema operacional como o Windows ou o Linux, um programa não executa sozinho, mas compartilha o processador e outros recursos do computador com outros programas e com o próprio sistema operacional. Esse compartilhamento é coordenado pelo sistema operacional, e o programador Scilab não tem qualquer controle sobre a forma como se dá este compartilhamento. A função `timer` procura calcular o tempo efetivo de processamento de um programa, mas com imprecisões que você já pôde apreciar. É por isso que uma mesma função, chamada com os mesmos parâmetros, pode apresentar diferenças de desempenho a cada execução.

[48] "The Prime Pages (prime number research, records and resources)", acessado 7 de março de 2011, http://primes.utm.edu/index.html.

Tabela 4: Tempos (segundos) gastos com a função MenorFator em um notebook e em um desktop

Primo	Notebook	Desktop
257	0,01560	0,03125
521	0,04680	0,01563
1031	0,03120	0,04688
2053	0,10920	0,01563
4099	0,17160	0,06250
8209	0,39000	0,21875
16411	0,68640	0,29688
32771	14,19609	0,76563
65537	29,17219	1,53125
131101	58,03237	2,93750
262147	11,94968	6,29688
524309	23,50935	11,21875
1048583	47,04990	24,54688
2097169	93,05460	49,60938

Outro experimento procura capturar o impacto do valor do número primo recebido como parâmetro sobre o tempo de execução da função `MenorFator`. A Tabela 4 mostra os resultados obtidos com diversos números primos, em experimentos realizados com dois computadores com velocidades distintas. O gráfico da Figura 328 resulta desta tabela, e ele torna clara a relação de linearidade entre o valor do número primo e o tempo consumido pela função `MenorFator`.

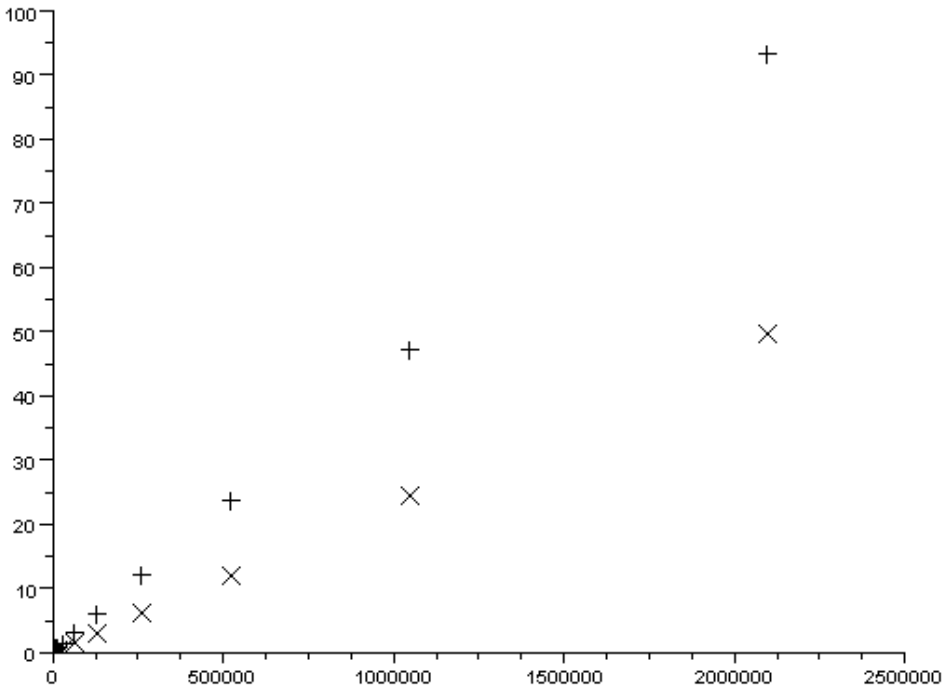


Figura 328: Tempo para fatoração em função do valor de números primos em dois computadores

A Ciência da Computação se preocupa exatamente em determinar a complexidade de um algoritmo por uma análise do algoritmo em si; experimentos naturalmente ajudam a formar uma intuição. Um exame da função **MenorFator** (Figura 316) nos teria permitido prever seu comportamento. Quando o número testado é primo, a função **modulo** é aplicada a todos os inteiros menores ou iguais a ele. Como aí se encontra todo o esforço computacional, poderíamos já ter previsto um crescimento linear do tempo de execução em função do valor do número primo examinado.

A constante de linearidade em si depende do computador onde a função **MenorFator** é executada. Para o notebook, o tempo gasto com um número primo p é aproximadamente igual a $p \times 4{,}49 \times 10^{-5}$ segundos; para o desktop, $p \times 2{,}31 \times 10^{-5}$ segundos, ou seja, o desktop é aproximadamente duas vezes mais rápido do que o notebook (não tome isso como uma afirmativa genérica; estamos falando do desktop e do notebook empregados nos experimentos).

Para poder comparar algoritmos, procura-se determinar a complexidade temporal de um algoritmo de forma independente da velocidade de um computador específico e de outros fatores que afetam o tempo de execução de um programa. Um algoritmo tem a sua complexidade conhecida quando conseguimos encontrar uma função que descreva a *ordem de crescimento* de seu tempo de execução com relação ao *tamanho* de sua entrada.

E qual é o tamanho da entrada da função **MenorFator**? Seu único parâmetro de entrada é um número, que, para fins de análise do algoritmo, podemos supor ser representado como binário sem sinal com n bits. Nós vimos que, para o pior caso da execução de **MenorFator**, o tempo cresce linearmente com o *valor* do primo fatorado, que é próximo de 2^n, onde n é o número de bits necessários para a representação do primo.

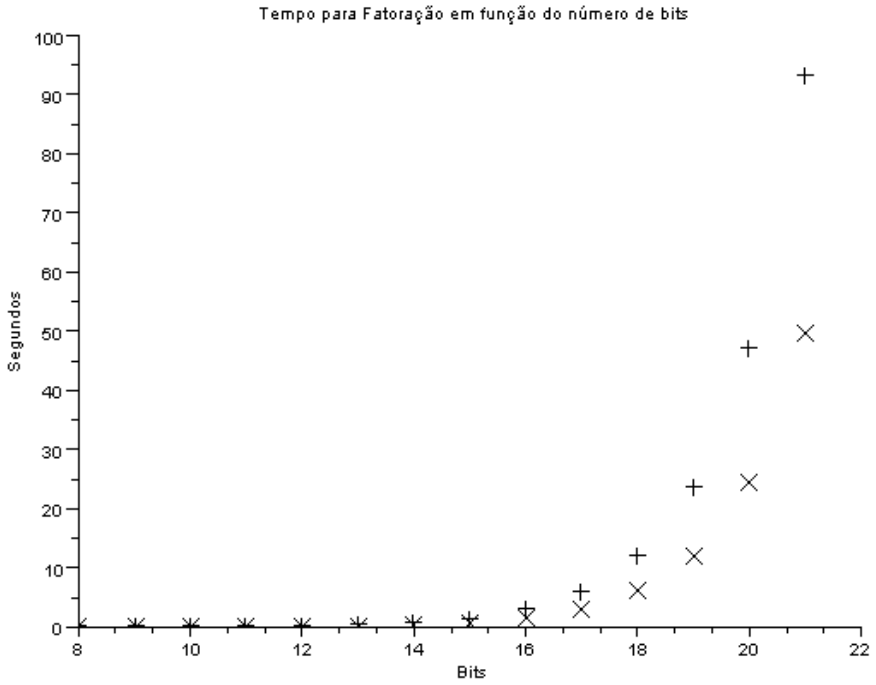


Figura 329: Tempos para fatoração em função do número de bits de um número primo em dois computadores

Se n é o tamanho da entrada de um problema, nós dizemos que uma função $g(n)$ caracteriza a complexidade de um algoritmo quando seu tempo de execução é limitado por $g(n)$ multiplicado por uma constante. Nós dizemos que sua complexidade é da ordem de $g(n)$, o que é escrito $T(n) = O(g(n))$ (pronuncia-se "Ó de $g(n)$"). No caso do algoritmo utilizado pela função **MenorFator**, temos $T(n) = O(2^n)$, e por isso dizemos que o algoritmo tem

complexidade exponencial. A idéia aqui é ter informação útil sobre o desempenho de um algoritmo, que não dependa da velocidade de computadores específicos. A constante de multiplicação serve para isso: ela absorve pequenas perturbações na execução, e pode incorporar diferenças de desempenho dos computadores.

9.3 Melhorando a Fatoração

Vamos agora considerar a função **MenorFator2**, mostrada na Figura 330, que implementa um algoritmo melhor para a fatoração de um número. Este novo algoritmo se baseia na observação de que, se d é um divisor de um inteiro positivo p, é porque existe d' tal que $d \times d' = p$. Se $d = d'$, $d = \sqrt{p}$ e p é um quadrado perfeito; se $d \neq d'$, ou bem $d < \sqrt{p}$ ou bem $d' < \sqrt{p}$. Isto significa que só precisamos testar a divisibilidade para os inteiros menores ou iguais à raiz quadrada de p; se neste intervalo não encontrarmos nenhum divisor, já poderemos concluir que p é primo.

```
function p = MenorFator2(n)
  limite = int(sqrt(n));
  p = 2;
  while modulo(n,p) <> 0 & p <= limite
    p = p + 1;
  end
  if modulo(n,p) <> 0 then
    p = n;
  end
endfunction
```

Figura 330: A função MenorFator2

Uma análise direta da função **MenorFator2** mostra que são realizadas chamadas da função módulo em número igual à raiz quadrada do valor do número primo sendo fatorado. Portanto, sua complexidade é $T(n) = O(\sqrt{2^n}) = O(2^{n/2})$. Experimentamos a função **MenorFator2** para verificar fatorar 2750159, o maior primo presente no arquivo 2000000primos.txt, e o tempo gasto no desktop foi de 0,047 segundos, enquanto que a função **MenorFator** gastou 88,360 segundos, uma diferença enorme de desempenho. Quando $n = 10$, a função **modulo** é chamada aproximadamente 1024 vezes pela **MenorFator**, e somente 32 vezes pela **MenorFator2**. Quando $n = 20$, a função **modulo** é chamada aproximadamente $2^{20} = 1.048.576$ vezes pela **MenorFator**, e 1024 vezes pela **MenorFator2**. Ao passar o número de bits da entrada de 10 para 20, a **MenorFator** demora 1024 vezes mais, enquanto a **MenorFator2** demora 32 vezes mais.

Função de complexidade	Tamanho da Instância do Problema					
	10	20	30	40	50	60
n	0,00001 segundos	0,00002 segundos	0,00003 segundos	0,00004 segundos	0,00005 segundos	0,00006 segundos
n^2	0,0001 segundos	0,0004 segundos	0,0009 segundos	0,0016 segundos	0,0025 segundos	0,0036 segundos
n^3	0,001 segundos	0,008 segundos	0,027 segundos	0,064 segundos	0,125 segundos	0,216 segundos
n^5	0,1 segundos	3,2 segundos	24,3 segundos	1,7 minutos	5,2 minutos	13,0 minutos
2^n	0,001 segundos	1,0 segundos	17,9 segundos	12,7 dias	35,7 anos	366 séculos
3^n	0,059 segundos	58 minutos	6,5 anos	3855 séculos	2×10^8 séculos	$1{,}3 \times 10^{13}$ séculos

Figura 331: Quadro comparativo de funções de complexidade

A Figura 331, extraída do livro de Garey e Johnson[49], nos ajuda a formar uma idéia do que esperar do desempenho de algoritmos com funções de complexidade exponencial quando aplicados a problemas grandes.

Maior instância que um computador resolve em 1 hora			
Função de complexidade	Computador Atual	Computador 100x mais rápido	Computador 1000x mais rápido
n	N	100 N	1000 N
n^2	M	10 M	31,6 M
n^3	Z	4,64 Z	10 Z
n^5	W	2,5 W	3,98 W
2^n	X	X + 6,64	X + 9,97
3^n	Y	Y + 4,19	Y + 6,29

Figura 332: Efeito do aumento da velocidade de computadores sobre o tamanho dos problemas resolvíveis

Computadores de um mesmo preço dobram de velocidade em menos de dois anos, mas algoritmos com funções de complexidade exponencial são relativamente pouco afetados. O efeito de termos computadores 100 ou 1000 vezes mais rápidos que os atuais sobre algoritmos com funções de complexidade exponencial está mostrado na Figura 332, que também foi retirada da referência[50]. Ali vemos que se hoje um computador resolve em uma hora um problema de tamanho, digamos, 200, usando um algoritmo $O(2^n)$, com um computador 1000 vezes mais rápido conseguiremos resolver um problema de tamanho ~210.

[49] M. R. Garey e D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, First Edition (W. H. Freeman, [s.d.]).
[50] Ibidem.

9.4 Pesquisa

Vamos agora estudar um problema clássico da Ciência da Computação, que é a pesquisa para verificar se um elemento procurado existe em uma tabela. Extensões deste problema fazem parte do nosso dia a dia, em máquinas de busca como Google e Yahoo, ou na localização de uma palavra em um arquivo. Vamos examinar aqui dois algoritmos: a pesquisa seqüencial e a pesquisa binária. Como um exemplo nós vamos utilizar algoritmos de pesquisa para testar a primalidade de um número, a tabela sendo composta pelos números primos presentes no arquivo 200000primos.txt. Obviamente isto só funciona para números menores ou iguais ao maior número presente no arquivo, 2750159.

A especificação do problema que iremos resolver de duas maneiras distintas é:

Faça um programa que:

- *Leia o arquivo 200000primos.txt, que contém os primeiros 200000 números primos;*
- *Leia repetidamente números inteiros e, para cada número lido, verifique se o número é primo pesquisando por ele na tabela;*
- *O programa deve parar quando o número lido for 0 (zero).*

9.5 Pesquisa Seqüencial

A Figura 333 mostra o programa **VerificaPrimos3.sci**, onde podemos notar que:

- Os primeiros comandos fazem a leitura da tabela de números primos;
- O programa apresenta a nossa velha conhecida estrutura de repetição controlada pelo usuário;
- A verificação efetiva da primalidade foi deixada para a função **Primo3**, que tem como parâmetros de entrada o número digitado pelo usuário e a tabela lida do arquivo.

```
// Programa para deteção de números primos
exec("Primo3.sci")
exec("seqSearch.sci")

arqTab = uigetfile("*.txt",pwd(),"Arquivo com Tabela");
tabPrimos = fscanfMat(arqTab);

n = input("n = ");
while n >= 2
  timer(); eh_Primo = Primo3(n,tabPrimos); tempoGasto = timer();
  // Imprime o resultado
  printf("\nTempo gasto = %g segundos", tempoGasto);
  if eh_Primo then
    printf("\nO número %d é primo!\n\n",n);
  else
    printf("\nO número %d não é primo!\n\n", n)
  end
  n = input("n = ");
end
```

Figura 333:O programa VerificaPrimos3.sci, que utiliza pesquisa seqüencial

A função **Primo3** é apenas um envelope sobre uma função de pesquisa seqüencial, como mostra a Figura 334.

```
function ePrimo = Primo3(n,tabela)
  ePrimo = seqSearch(n,tabela) ~= -1;
endfunction
```

Figura 334: A função Primo3

A função **seqSearch** mostrada na Figura 335 implanta uma pesquisa seqüencial. Ali podemos observar que:

- O vetor **table** é examinado sequencialmente a partir de sua primeira posição.
- O loop de pesquisa pára por um de dois motivos: quando o limite superior da tabela for atingido, ou quando a chave for encontrada.
- Após a saída do loop é feito um teste para se saber por qual motivo o loop while terminou.
- Se a chave procurada não consta da tabela, o parâmetro de saída **p** recebe o valor -1, uma convenção útil para quem chama a função, como a função **Primo3**.

```
function p = seqSearch(key,table)
  i = 1;
  while i <= length(table) & table(i) ~= key
    i = i+1;
  end
  if i <= length(table) then
    p = i;
  else
    p = -1;
  end
endfunction
```

Figura 335: A função seqSearch para pesquisa seqüencial

Quanto à complexidade da pesquisa seqüencial, não é difícil ver que se n é o tamanho da tabela, o número de comparações com a chave feito em uma pesquisa por um elemento presente na tabela varia entre 1 e n; pesquisas por elementos que não constam da tabela (o que constitui o pior caso) consomem sempre n comparações com a chave. Se considerarmos todas as chaves presentes na tabela como tendo a mesma probabilidade de serem pesquisadas, o algoritmo fará em média $n/2$ comparações por pesquisa por chave constante da tabela.

O número de comparações cresce portanto linearmente com o tamanho da tabela, e nós dizemos que o algoritmo de pesquisa seqüencial é $O(n)$, ou seja, da ordem de n, ou ainda, dizemos que a pesquisa seqüencial tem complexidade linear.

9.6 Pesquisa Binária

Quando uma tabela tem suas entradas dispostas em ordem crescente ou decrescente nós podemos usar um algoritmo muito mais eficiente para a pesquisa, e que se assemelha ao método como nós humanos procuramos palavras em um dicionário. A primeira comparação é feita não com o primeiro elemento da tabela, mas com o elemento no meio da tabela. Supondo que os elementos da tabela estão em ordem crescente, se a chave procurada for menor que o elemento no meio da tabela, podemos restringir a pesquisa à metade inferior da tabela, pois a parte superior só contém elementos maiores do que a chave procurada. Da mesma forma, se a chave procurada for maior que o elemento no meio da tabela, podemos restringir a pesquisa à metade superior da tabela.

O método é reaplicado à parte restante da tabela, e continua até que ou a chave é encontrada, ou a parte da tabela em consideração tem tamanho igual a 0, situação em que podemos

concluir que a chave não consta da tabela. A denominação de pesquisa binária vem do fato da divisão do tamanho do problema por 2 a cada passo do algoritmo.

```
function p = BinarySearch(key,table,low,high)
  if high < low then
    p = -1;
  else
    m = int((low+high)/2);
    if key == table(m) then
      p = m;
    else
      if key < table(m) then
        p = BinarySearch(key,table,low,m-1);
      else
        p = BinarySearch(key,table,m+1,high);
      end
    end
  end
endfunction
```

Figura 336: A função recursiva BinarySearch

A Figura 336 mostra uma implementação direta da pesquisa binária como uma função recursiva.

```
function position = binSearch(key,table)
  low = 1;  high = length(table);
  while high - low > 1
    m = int((high+low)/2);
    if key >= table(m) then
      low = m;
    end
    if key <= table(m) then
      high = m;
    end
  end
  if key == table(high) then
    position = high;
  else
    if key == table(low) then
      position = low;
    else
      position = -1;
    end
  end
endfunction
```

Figura 337: A função binSearch

A função `binSearch` (Figura 337) é uma implementação não recursiva em Scilab do algoritmo de pesquisa binária, onde podemos observar que:

- a função utiliza dois ponteiros, `low` e `high`, que indicam a cada passo qual parte da tabela que pode conter a chave procurada. A condição `low <= high` é mantida em todas as alterações de valores dessas variáveis.

- a cada passagem do loop o elemento comparado com a chave buscada está na posição `m = int(high+low)/2` .
- o ponteiro `low` é tratado de forma a manter sempre a afirmativa "se `key < table(low)`, então `key` não consta da tabela", que é valida inicialmente, e que permanece válida a cada atualização de `low`;
- o ponteiro `high` cumpre papel similar, mantendo sempre válida a afirmativa "se `key > table(high)`, então `key` não consta da tabela" ;
- quando o loop termina, a parte da tabela que pode conter um elemento igual à chave está entre `low` e `high` e, pela condição de término do loop e pela relação entre `low` e `high`,`0 <= high - low < = 1`.
- os testes que sucedem ao loop permitem decidir se a tabela contém ou não um elemento igual à chave procurada.

A análise da complexidade do pior caso da pesquisa binária é simples. A cada passo o tamanho da parte da tabela que pode conter a chave é dividido por 2, e o algoritmo termina quando o tamanho desta parte é igual a 1. Se a tabela tem n elementos, teremos no pior caso, $\lceil \log_2 n \rceil$ comparações, onde $\lceil x \rceil$ indica a função *teto*, que mapeia um número real x no menor inteiro maior ou igual a x. No caso da tabela armazenada no arquivo 200000primos.txt, temos no pior caso $\lceil \log_2 200.000 \rceil = \lceil 17{,}60964 \rceil = 18$ comparações para completar a pesquisa. Compare isso com o pior caso da pesquisa seqüencial, que pode necessitar de 200.000 passos para terminar. Pior: se a tabela dobrar de tamanho, passando a ter 400.00 elementos, o algoritmo de pesquisa binária irá passar a necessitar de 19 comparações, uma a mais, enquanto que o algoritmo de pesquisa seqüencial poderia necessitar de 400.000 passos.

Por estes argumentos alguém poderia pensar que, em casos em que temos uma tabela ordenada, a opção pela pesquisa binária em detrimento da pesquisa seqüencial é uma escolha óbvia, mas nem sempre é este o caso. A pesquisa seqüencial é mais simples, mais fácil de programar, e menos propensa a erros de programação. E muitas vezes trabalhamos com tabelas pequenas, onde a diferença de desempenho entre os dois algoritmos não é importante. Ao programar, nunca se esqueça do *kiss principle*, um acrônimo para uma frase em inglês que todos os bons programadores dizem a si próprios todos os dias: *keep it simple, stupid*. Ou seja, só complique quando inevitável.

10 Ordenação

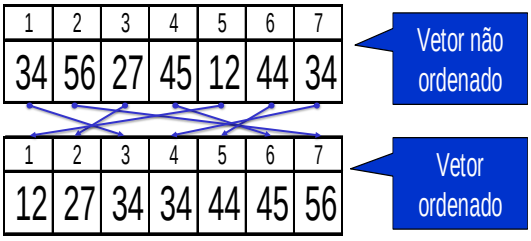


Figura 338: Ordenação de um vetor

Outro problema clássico da Ciência da Computação é a ordenação, que consiste em, dado um vetor A, obter outro vetor com os mesmos elementos de A, dispostos em ordem crescente (ou decrescente), como mostra a Figura 338. Uma outra forma de se ver o efeito de uma operação de ordenação está ilustrada na Figura 339.

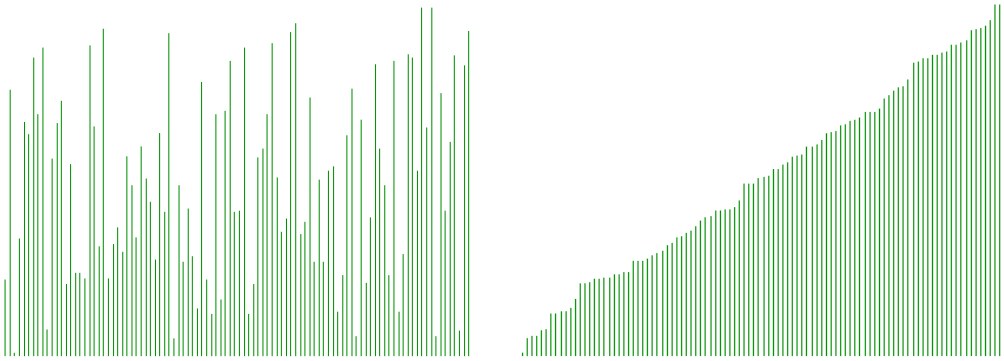


Figura 339: Gráfico dos valores de um conjunto desordenado e após sua ordenação

Existem muitos algoritmos para ordenação; nós veremos três deles aqui, conhecidos por algoritmos de ordenação por seleção e troca, por intercalação e por intercalação.

10.1 Seleção e Troca

O primeiro algoritmo que iremos estudar é conhecido como o método de Seleção e Troca (em inglês, *Select Sort*), por motivos que se tornarão óbvios. Vamos começar apresentando o cabeçalho de uma função que iremos desenvolver para implantar esse algoritmo, mostrado na Figura 340.

```
function sA = SelectSort(A)
  // Constrói o vetor sA com os
  // mesmos elementos do vetor A
  // dispostos em ordem crescente.
endfunction
```

Figura 340: Cabeçalho de uma função de ordenação

Isso já nos permite desenvolver um programa para testar a função, mostrado na Figura 341.

```
exec("SelectSort.sci");
a = int(10*rand(1,4))
aa = SelectSort(a)
b = int(10*rand(1,6))
bb = SelectSort(b)
c = int(10*rand(1,9))
cc = SelectSort(c)
```

Figura 341: O programa SelectSort_teste.sce

O programa `SelectSort_teste.sce` é bastante simples, e nos permite verificar por inspeção visual a correção da função por testes com pequenos vetores randômicos. Ele gera 3 pequenos vetores randômicos que são passados como parâmetros de entrada para a função. Repare na ausência dos "`;`" em quase todos os comandos; queremos tirar proveito do eco automático do Scilab para a impressão dos vetores antes e depois da ordenação.

O algoritmo de ordenação por seleção e troca é também desenvolvido por um raciocínio indutivo:

- Suponhamos que as `k-1` primeiras posições do vetor `A` já contenham elementos em suas posições finais;
- Selecionamos o elemento de menor valor entre `A(k)` e `A(length(A))` ;
- Trocamos o valor deste menor elemento com o valor em `A(k)`. Com isso teremos mais um elemento em sua posição final, e podemos fazer `k = k+1`.

O algoritmo se inicia com `k` igual a 1, o que torna vazia a porção do vetor com elementos em suas posições finais.

Na Figura 342 nós vemos uma ilustração deste método:

- No primeiro passo o menor elemento entre 1 (igual a `k`) e 7 (comprimento do vetor) tem o valor 12, e se encontra na posição 7. Os valores nas posições 1 e 7 são trocados, e `k` passa a valer 2.
- No segundo passo, o menor elemento entre 2 (igual a `k`) e 7 tem o valor 27, e se encontra na posição 3. Ele é trocado com o elemento na posição 2, e `k` passa a valer 3.

Desta forma o algoritmo progride, e termina quando a penúltima posição do vetor recebe o seu valor final – o que significa que a última posição também estará corretamente preenchida.

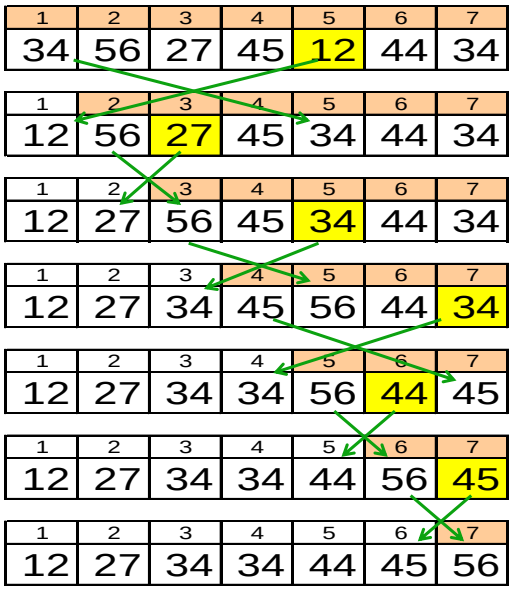


Figura 342: Ordenação por Seleção e Troca

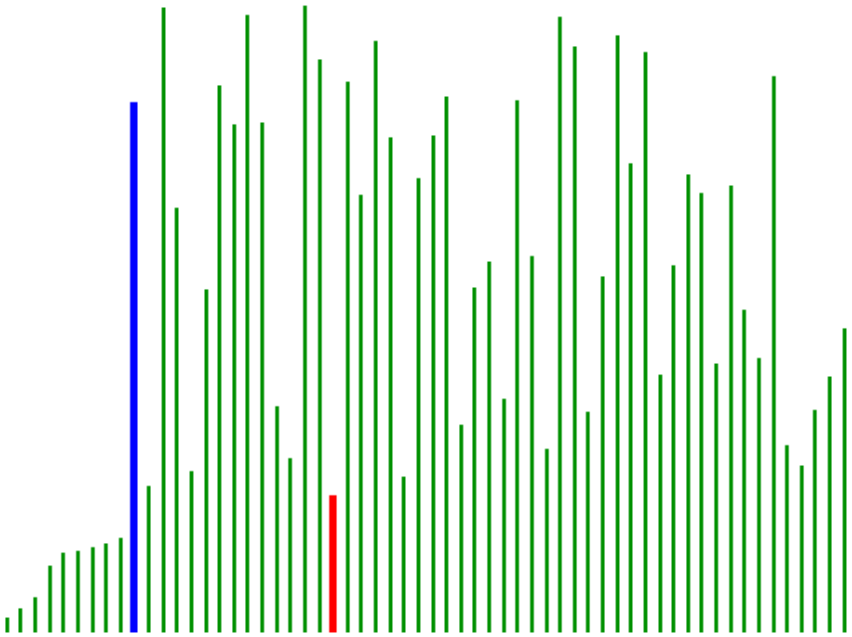


Figura 343: Seleção de elementos para troca de posição: em azul, o elemento na primeira posição da parte ainda não ordenada e, em vermelho, o menor elemento na parte não ordenada

Já temos condições de dar um primeiro refinamento à função `SelectSort`, que mostramos na Figura 344.

```
function sA = SelectSort(A)
  for k = 1:length(A)-1
    // Seleciona a posição entre A(k) e
    // A(length(A)) que contém o menor valor

    // Troca o valor de A(k) com o valor na
    // posição selecionada.
  end
  sA = A;
endfunction
```

Figura 344: Primeiro refinamento da função OrdenaPorSeleção

Prosseguindo no refinamento, vamos atacar inicialmente o problema da troca de valores entre duas posições do vetor `A`. Trocar os valores de duas variáveis `x` e `y` não é uma operação óbvia. Se fizermos `x = y; y = x`, o valor antigo de `x` que queríamos armazenar em `y` é perdido. Se fizermos `y = x; x = y`, teremos o problema inverso. A solução é o uso de uma variável temporária (adjetivo empregado para variáveis cuja utilidade tem um caráter destacadamente local e auxiliar) `temp`, e fazer `temp = x; x = y; y = temp`, o que nos dá a solução desejada. Muito simples, depois que sabemos.

Para a etapa de seleção do menor elemento, nós já desenvolvemos um loop com função parecida (veja a Figura 254), que encontra o menor valor presente em um vetor. Podemos aproveitar o seu código, adaptando-o aos requisitos que temos agora. Precisamos de uma função, que vamos chamar de `MinimoPos`, que:

- procure o menor valor não em todo o vetor de entrada, mas em parte dele, e
- informe além do menor valor, a posição onde foi encontrado.

```
function [m,im] = MinimoPos(A,low,high)
  // Encontra o menor valor
  // presente no vetor A entre as
  // posições low e high, e informa
  // sua posição no vetor
  m = A(low);
  im = low;
  for k = low+1:high
    if m > A(k)
      m = A(k);
      im = k;
    end
  end
endfunction
```

Figura 345: A função SelecionaMenor

Como você pode reparar na Figura 345, as modificações introduzidas memorizam na variável **im** a posição onde o mínimo corrente foi encontrado (nas situações em que **m** é atualizada), restringem o espaço da busca apenas aos elementos de **A** com índices entre os parâmetros de entrada **low** e **high**, e acrescentam um parâmetro de saída.

Se a função **Minimo** mereceu a construção de um programa para seu teste, é uma boa idéia fazer o mesmo para a função **MinimoPos**, que é um pouco mais complicada. Você poderia pensar que, indiretamente, o programa **OrdenaPorSelecao_teste** já o faria, mas construir um testador independente tem a grande vantagem de simplificar o contexto de uso da função **MinimoPos**.

```
// Programa que testa a função MinimoPos
exec("MinimoPos.sci");
exec("PrintMatrix.sci");

a = int(100*rand(1,10));
PrintMatrix("A",a);
inicio = input("Inicio = ");
fim = input("Fim = ");
while inicio > 0
  [m im] = MinimoPos(a,inicio,fim)
  printf("Menor valor entre %d e %d = %d, na posição %d",...
  inicio,fim,m,im);
  inicio = input("Inicio = ");
  fim = input("Fim = ");
end
```

```
function PrintMatrix(Label, M);
  printf("\n%s = [",Label);
  for i = 1:length(M)
    printf("%3d",M(i));
  end
  printf("]")
endfunction
```

Figura 346: O programa MinimoPos_teste e a função PrintMatrix

A Figura 346 mostra o programa **MinimoPos_teste.sce**, juntamente com uma função auxiliar **PrintMatrix**, cuja utilidade é simplesmente a impressão de um vetor em um formato mais agradável do que o padrão do Scilab. Este programa gera um vetor com 10

elementos aleatórios, e permite que o usuário repetidamente escolha um ponto inicial para a seleção da posição com o menor elemento a partir do ponto inicial. A Figura 347 mostra uma saída deste programa.

```
A = [ 95 12 73  8 80  2 65 74 21 97]
Inicio = 1

Fim = 3
Menor valor entre 1 e 3 = 12, na posição 2
Inicio = 2

Fim = 10
Menor valor entre 2 e 10 = 2, na posição 6
Inicio = 0

Fim = 0
```

Figura 347: Uma saída do programa MinimoPos_teste

Com a confiança adquirida sobre a função `MinimoPos` nós podemos chegar ao refinamento final da função `SelectSort`, mostrado na Figura 348.

```
function sA = SelectSort(A)
  for k = 1:length(A)-1
    // Seleciona a posição entre A(k) e
    // A(length(A)) que contém o menor valor
    [Min iMin] = MinimoPos(A,k,length(A));
    // Troca os valores de A(k) com o valor
    // na posição selecionada.
    temp = A(k);
    A(k) = A(iMin);
    A(iMin) = temp;
  end
  sA = A;
endfunction
```

Figura 348: A função SelectSort

Vamos agora examinar a complexidade deste algoritmo. Para ordenar um vetor de tamanho n, o primeiro passo do algoritmo de seleção e troca realiza $n-1$ comparações; o segundo, $n-2$; o terceiro, $n-3$, e assim por diante, até chegar ao último passo, quando é feita uma única comparação. Podemos concluir que o número de comparações realizado é dado por

$$comp = 1 + 2 + \cdots + (n-1) = \frac{(n-1)n}{2} = \frac{n^2 - n}{2}$$

Para n suficientemente grande, o número de comparações se aproximará de $\frac{1}{2}n^2$. Ou seja, o número de comparações necessárias para a execução do algoritmo cresce com o quadrado do tamanho do vetor, e portanto o algoritmo de ordenação por seleção e troca é $O(n^2)$.

A Figura 349 mostra um gráfico com medidas de desempenho obtidas para o método de seleção e troca em dois computadores. Você pode reparar que para ordenar um vetor com 5000 elementos o tempo gasto pelo computador mais rápido já é significativo, da ordem de 1 minuto.

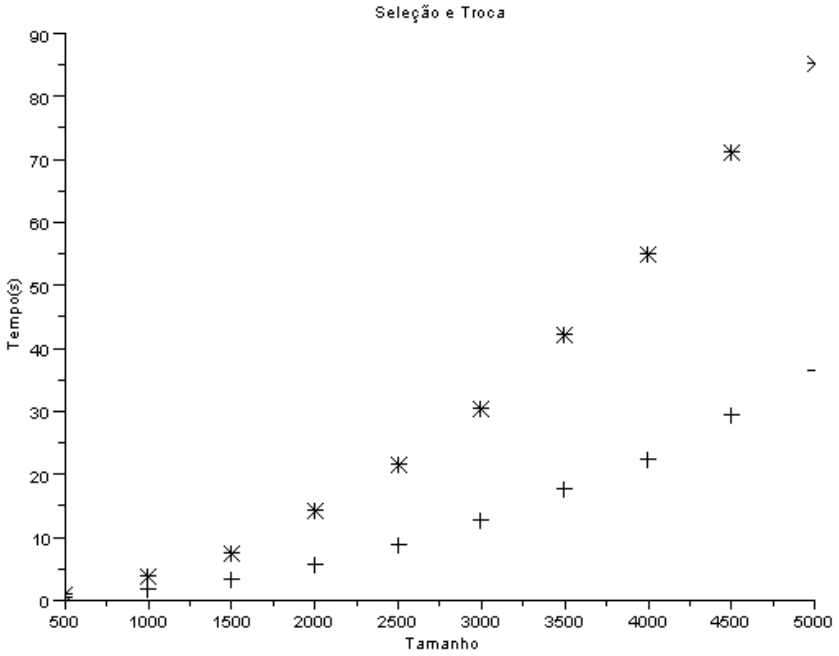


Figura 349: Tempos medidos para ordenação por seleção e troca de vetores aleatórios em um notebook (*) e em um desktop (+)

10.2 Intercalação (MergeSort)

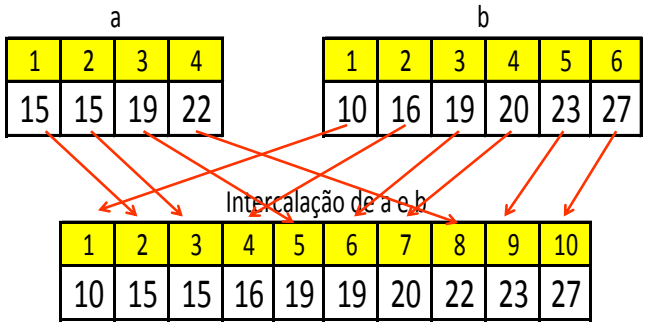


Figura 350: Uma operação de intercalação

Intercalação (em inglês, *merge*) é o nome dado ao processo de produção de um vetor ordenado a partir de dois outros já ordenados, como mostrado na Figura 350. O algoritmo de ordenação por intercalação consiste em dividir o vetor de entrada em duas partes, ordenar cada uma delas separadamente, e depois obter o vetor ordenado realizando uma operação de intercalação. A ordenação por intercalação é aplicada recursivamente a cada parte, a não ser que a parte a ser ordenada seja de tamanho 1, quando a recursão termina pois a parte já está trivialmente ordenada.

A Figura 351 mostra um exemplo de ordenação por intercalação de um vetor com 16 posições. Na parte superior da figura as pequenas setas indicam divisões de uma parte do vetor; na parte inferior (em vermelho) as pequenas setas indicam operações de intercalação de partes já ordenadas.

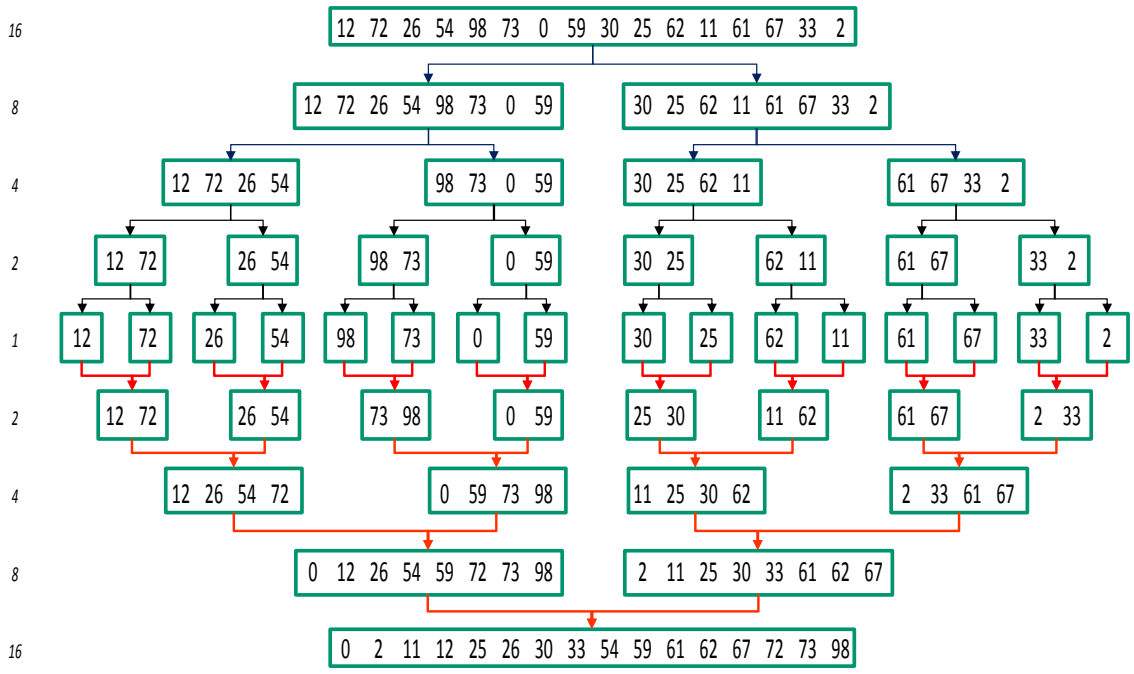


Figura 351: Exemplo de ordenação por intercalação com n = 16 = 2^4

Os pequenos números no lado esquerdo da figura indicam o tamanho de cada parte. Como 16 é uma potência de dois, a estrutura de divisões e intercalações é muito regular. A Figura 352 mostra um exemplo com um vetor de tamanho 10, quando nem sempre as duas partes resultantes de uma divisão (ou a serem intercaladas) têm o mesmo tamanho.

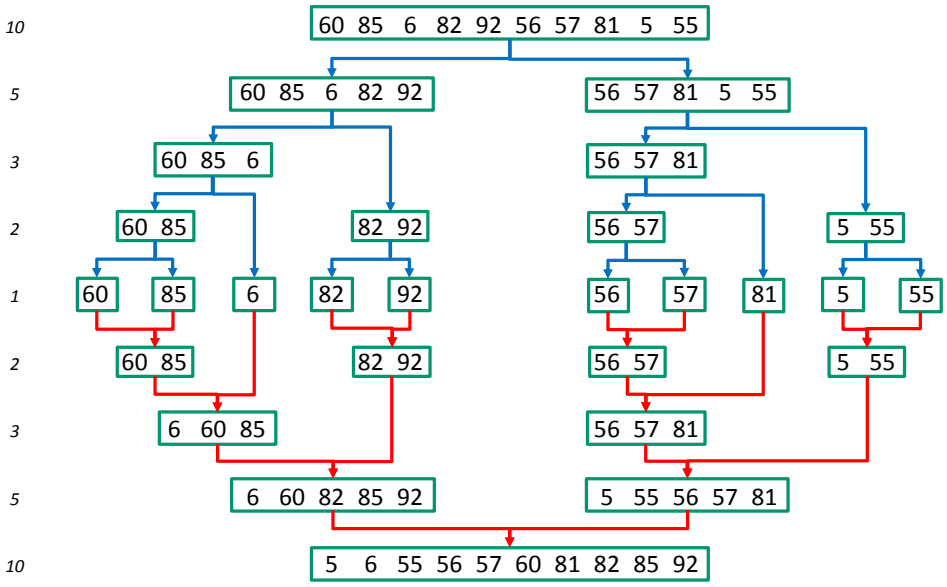


Figura 352: Exemplo de ordenação por intercalação com n = 10

O código da função `MergeSort` está mostrado na Figura 353. Você deve reparar que o código transcreve o algoritmo de forma bastante direta, e que pressupõe a existência de uma função `Merge` que realiza as intercalações.

```
function sA = MergeSort(A)
  if length(A) <= 1 then
    sA = A;
  else
    m = int((1+length(A))/2);
    sA = Merge(MergeSort(A(1:m)),...
               MergeSort(A(m+1:length(A))));
  end
endfunction
```

Figura 353: A função MergeSort

A codificação da função **Merge** (Figura 354) é também bastante intuitiva. Você deve reparar que:

- **pA**, **pB** e **pM** são ponteiros que indicam as posições em foco nos vetores fonte e no vetor resultado;
- O primeiro loop realiza a intercalação propriamente dita, a cada passo colocando em **M(pM)** o menor entre **A(pA)** e **B(pB)**, e avançando adequadamente **pA** ou **pB**; este loop termina quando **pA** ou **pB** atinge o limite do seu vetor;
- O segundo loop só é efetivamente executado quando a saída da fase de intercalação se deu pelo esgotamento dos elementos de **B** e consiste em copiar os elementos restantes de **A** em **M**;
- Da mesma forma, o terceiro loop só é efetivamente executado quando a saída da fase de intercalação se deu pelo esgotamento dos elementos de **A**, e consiste em copiar os elementos restantes de **B** em **M**.

```
function M = Merge(A,B)
  pA = 1; pB = 1; pM = 1
  while pA <= length(A) & pB <= length(B)
    if A(pA) <= B(pB) then
      M(pM) = A(pA);
      pA = pA+1;
    else
      M(pM) = B(pB);
      pB = pB+1;
    end
    pM = pM+1;
  end

  // Esgota A
  while pA <= length(A)
    M(pM) = A(pA);
    pM = pM+1;
    pA = pA+1;
  end

  // Esgota B
  while pB <= length(B)
    M(pM) = B(pB);
    pM = pM+1;
    pB = pB+1;
  end
endfunction
```

Figura 354: A função Merge

É fácil ver que o número de operações elementares de uma operação de intercalação de dois vetores ordenados de tamanhos m e n é da ordem de $m + n$. Isso porque cada passo do algoritmo produz um elemento do vetor de saída, cujo tamanho é $m + n$. A complexidade da ordenação por intercalação pode ser inferida da Figura 355.

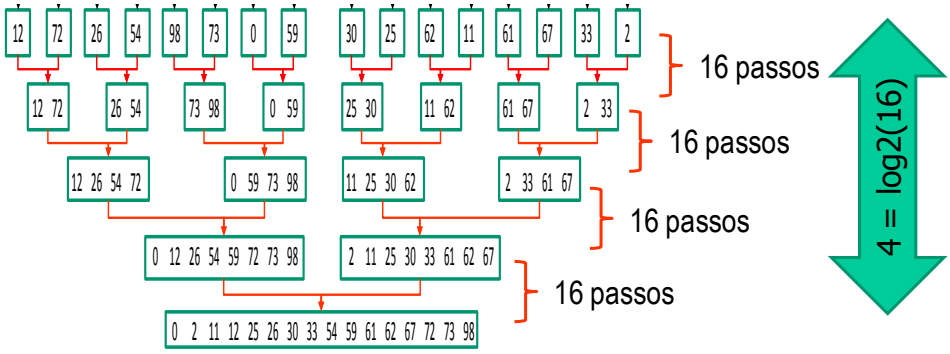


Figura 355: Passos para ordenação por intercalação

Ali vemos o processo ilustrado para um vetor cujo tamanho é uma potência de 2, mas para outros tamanhos, podemos considerar como limite superior a complexidade exigida para a menor potência de 2 que supere o tamanho do vetor. Vamos considerar apenas a fase de intercalação, ilustrada na parte inferior da figura, uma vez que as divisões são feitas com pouco trabalho computacional. Repare que:

- para a última intercalação realizada, que gera o vetor ordenado de tamanho n a partir de dois vetores ordenados de tamanho $n/2$, são feitas no n operações.
- para cada intercalação no penúltimo nível, que geram vetores ordenados de tamanho $n/2$ a partir de vetores de tamanho $n/4$, são feitas no $n/2$ operações; como temos duas intercalações neste nível, teremos também no máximo n comparações.
- o mesmo vale para todos os níveis anteriores; como temos $log_2(n)$ níveis, o número máximo de comparações realizadas pelo algoritmo de ordenação por intercalação é igual a $nlog_2(n)$, e o algoritmo é portanto $O(nlog_2 n)$.

Para se ter uma idéia da diferença de desempenho entre o algoritmo de seleção e troca e o algoritmo de intercalação, quando $n = 10$, $n^2 = 100$, e $nlog_2 n = 31{,}22$; quando $n = 1000$, $n^2 = 1.000.000$, e $nlog_2 n = 9965$, e por aí vai.

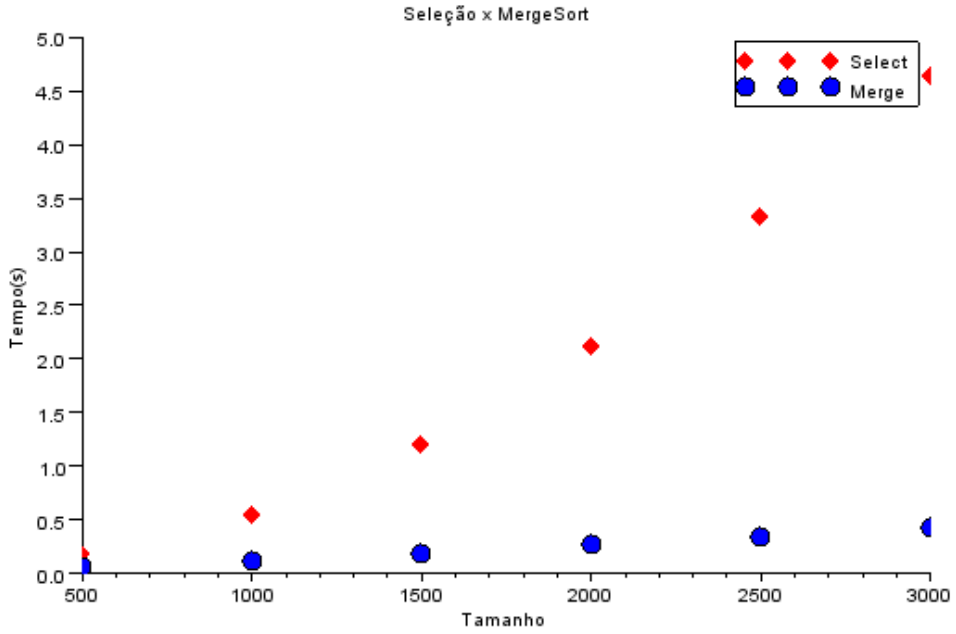


Figura 356: Comparação de desempenho SelectSort vs MergeSort

A Figura 356 mostra o resultado de testes de desempenho realizados com a função `MergeSort`, comparado com o desempenho da função `SelectSort`, A superioridade do MergeSort é gritante, e se acentua com o aumento do tamanho do vetor ordenado.

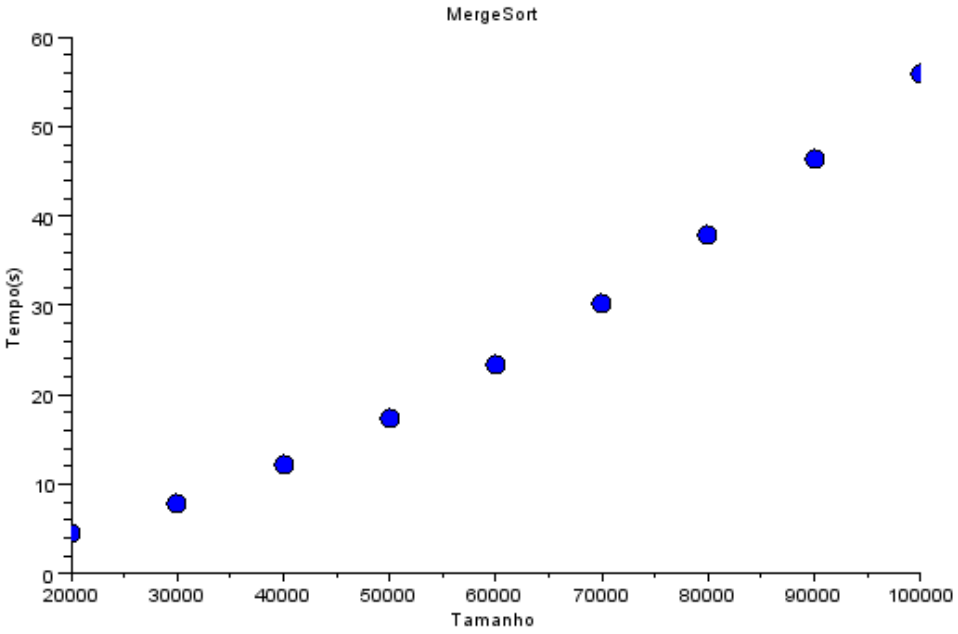


Figura 357: Gráfico de desempenho para a função MergeSort

Na Figura 357 nós vemos números de desempenho para o `MergeSort` obtidos com a ordenação de vetores bem maiores, para os quais esperar pelo `SelectSort` exigiria bastante paciência.

10.3 Partição (QuickSort)

Considere um vetor como o da Figura 358. Nós dizemos que este vetor está *particionado* porque sua parte esquerda, com índices de 1 a 4, só contém valores menores ou iguais a 100, e

sua parte direita, com índices de 5 a 9, só contém valores maiores ou iguais a 100. Se ordenarmos cada uma das partes de forma independente, o vetor completo estará ordenado, pois não há possibilidade de serem necessárias inversões de ordem entre elementos de diferentes partições. A Figura 359 mostra uma ilustração gráfica de um vetor particionado.

1	2	3	4	5	6	7	8	9
93	100	56	12	123	100	231	212	112

Figura 358: Um vetor particionado

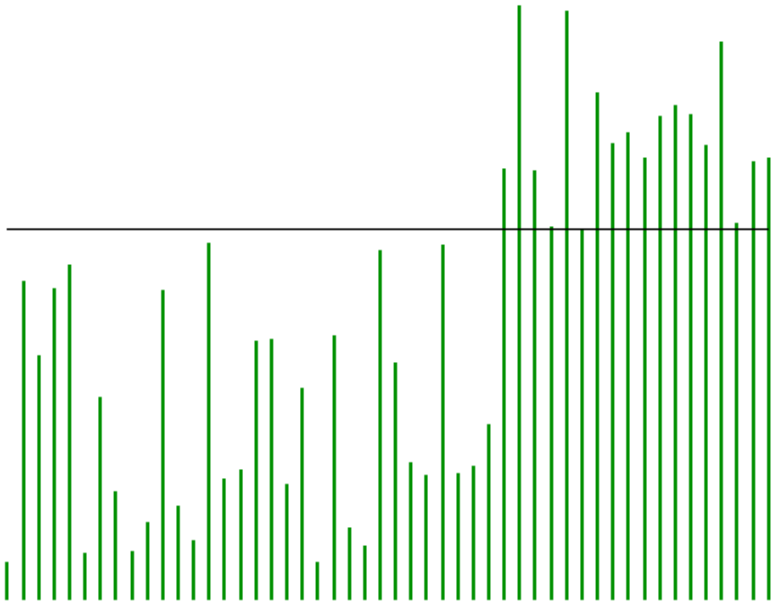


Figura 359: Visualização de um vetor particionado

O método que veremos agora para se ordenar um vetor utiliza uma operação de *partição* do vetor segundo um *pivô*, que é um valor igual a um dos elementos do vetor.

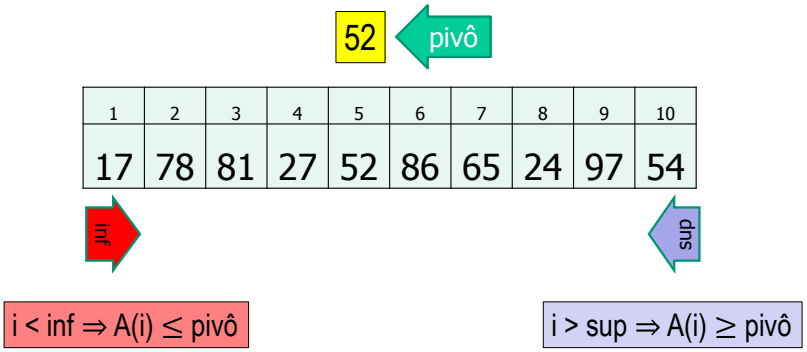


Figura 360: Início de uma operação de partição

A partição, operação aplicável somente a vetores com dois ou mais elementos, separa o vetor em três partes:

- a da esquerda, contendo somente elementos menores ou iguais ao pivô,
- a do meio, contendo somente elementos iguais ao pivô, e
- a da direita contendo somente elementos maiores ou iguais ao pivô.

É possível que algumas das partes resultantes seja vazia. O método prossegue aplicando-se recursivamente às partições da esquerda e da direita, até se conseguir partições de tamanho 1 que estão trivialmente ordenadas. Este é um dos algoritmos mais famosos da ciência da computação, tendo recebido o nome de *quicksort*.

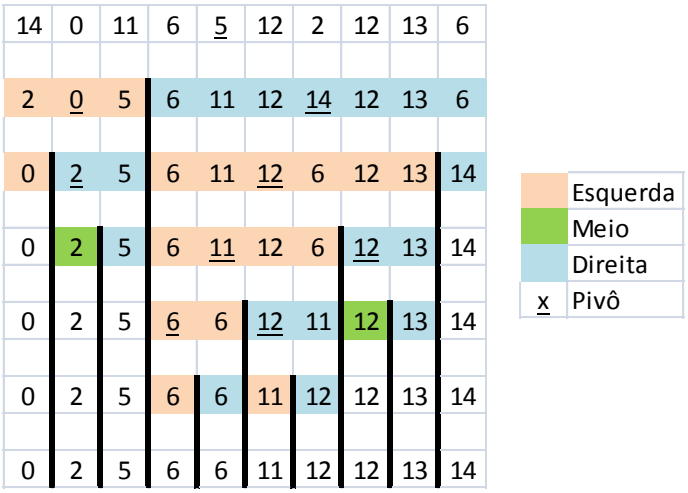


Figura 361: Exemplo de ordenação por partição (quicksort)

A Figura 361 ilustra o comportamento do algoritmo na ordenação de um vetor com 10 elementos. A cada partição o pivô escolhido está sublinhado.

```
function sA = quicksort(A)
  if length(A) <= 1 then
    sA = A;
  else
    [l,m,r] = partition(A);
    sA = [quicksort(l) m quicksort(r)];
  end
endfunction
```

Figura 362: A função quicksort, que implementa o algoritmo de ordenação por partição

Assim como o algoritmo de intercalação, a implantação do quicksort por meio de uma função recursiva é uma expressão direta do algoritmo, como mostra a Figura 362.

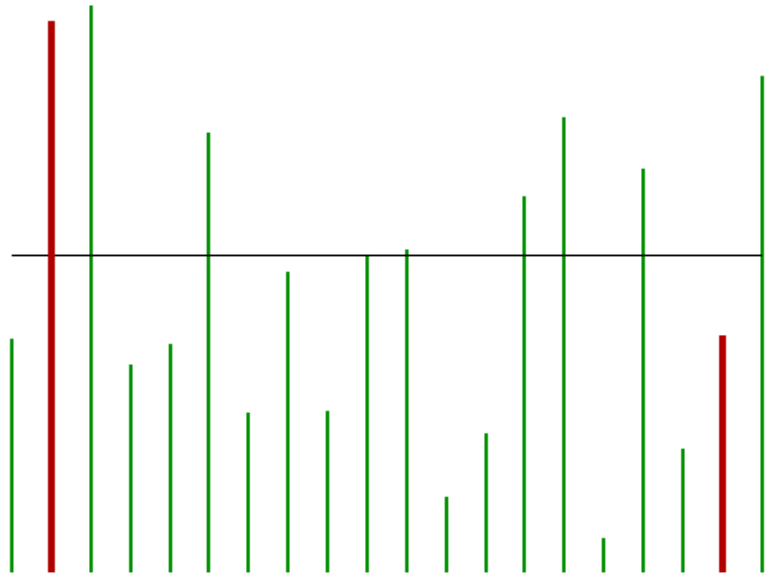


Figura 363: Elementos a serem trocados de posição durante o processo de partição. A barra horizontal mostra o valor do pivô.

As partições são feitas pela função `partition`, mostrada na Figura 364.

```
function [left,middle,right] = partition(A)
  pivot = A((1+int(length(A))/2));
  inf = 1;
  sup = length(A);
  while sup >= inf
    while A(inf) < pivot
      inf = inf+1;
    end
    while A(sup) > pivot
      sup = sup-1;
    end
    if sup >= inf then
      temp = A(inf); A(inf) = A(sup); A(sup) = temp;
      inf = inf+1; sup = sup-1;
    end
  end
  left = A(1:sup); middle = A(sup+1:inf-1);
  right = A(inf:length(A));
endfunction
```

Figura 364: A função partition

Diversos pontos são dignos de nota nesta função:

- O pivô é escolhido como o elemento posicionado ao meio do vetor `A`. Isto não é um requisito do algoritmo, que entretanto exige que o pivô seja um dos elementos de `A`. Algumas versões escolhem o primeiro elemento do vetor como pivô, enquanto outras sorteiam a sua posição;
- `inf` avança sempre para a direita, e `sup` para a esquerda; o loop principal pára quando estes dois ponteiros se cruzam;
- no primeiro loop interno, `inf` avança para a direita até encontrar um elemento com valor maior ou igual ao pivô;

- no segundo loop interno, **`sup`** avança para a esquerda até encontrar um elemento com valor menor ou igual ao pivô;
- os elementos encontrados nestes dois loops internos são trocados de posição, a não ser que **`inf`** e **`sup`** já tenham se cruzado;
- **`inf`** é mantido de forma tal que, a qualquer momento, todos os elementos à sua esquerda são menores ou iguais ao pivô. Repare que isto é válido inicialmente, pois não existe nenhum elemento à sua esquerda, e que é mantido válido por todos os comandos;
- **`sup`** é mantido de forma tal que, a qualquer momento, todos os elementos à sua direita são maiores ou iguais ao pivô; a argumentação para esta afirmativa é análoga à empregada para a variável **`inf`**;

Não é difícil ver que a função **`partition`** realiza s comparações para dividir uma partição de tamanho s. O comportamento do algoritmo de ordenação por partição é fortemente dependente das escolhas de pivô. No melhor caso, os pivôs escolhidos são tais que cada partição divide o vetor ao meio e, por motivos similares aos colocados na análise da complexidade do algoritmo de ordenação por intercalação, o quicksort é $O(nlog_2 n)$. No pior caso, o pivô é tal que cada partição produz uma parte com um único elemento e outra com $n-1$ elementos, e o número de comparações é $O(n^2)$, tão ruim como a ordenação por seleção e troca.

Os piores casos do quicksort são geralmente pouco prováveis, e seu desempenho em casos como os nossos testes com vetores aleatórios é significativamente superior ao do mergesort.

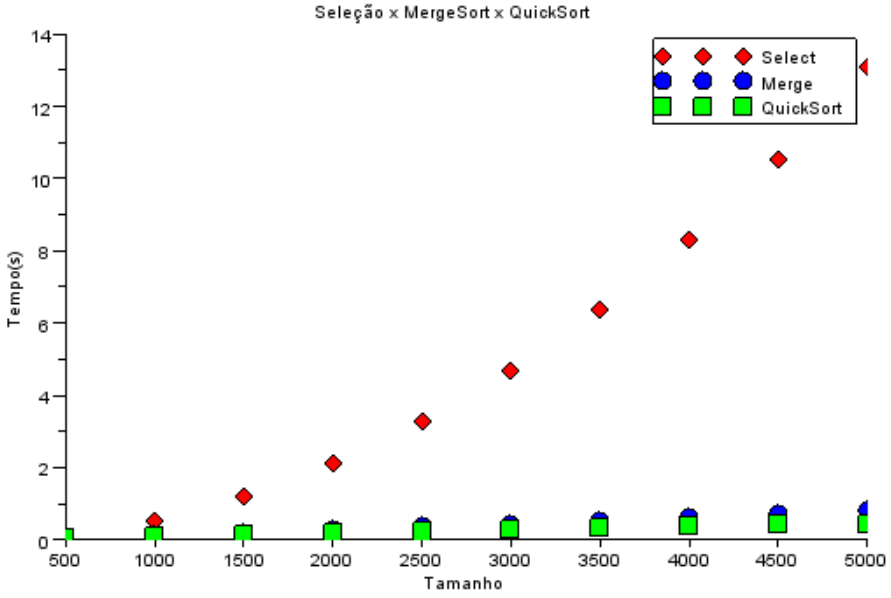


Figura 365: Comparação entre SelectSort, MergeSort e QuickSort

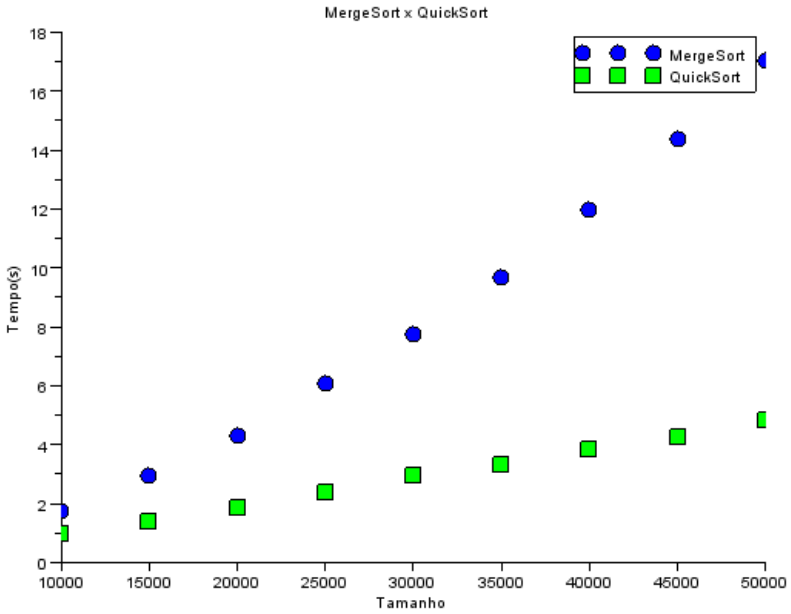


Figura 366: Comparação entre MergeSort e QuickSort

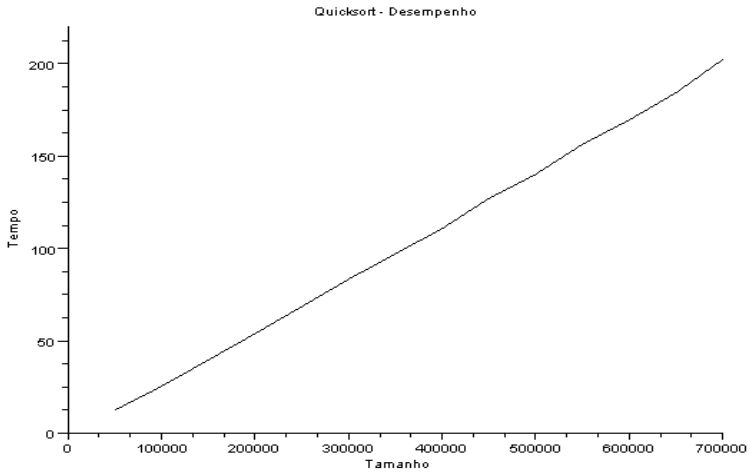


Figura 367: Teste de desempenho do quicksort

Na Figura 367 nós mostramos os resultados obtidos com vetores randômicos de até 700.000 elementos, que já atingem o limite de memória disponível no Scilab. Repare que em 60 segundos o MergeSort foi capaz de ordenar 50.000 elementos, e que, neste mesmo tempo, o Quicksort ordena 200.000 elementos. Os resultados são tão bons que o crescimento da curva de tempo parece linear. Como sabemos que no melhor caso o tempo deveria crescer com $n\ log(n)$, fizemos outra curva para nos certificarmos da correção do nosso experimento, mostrada na Figura 368.

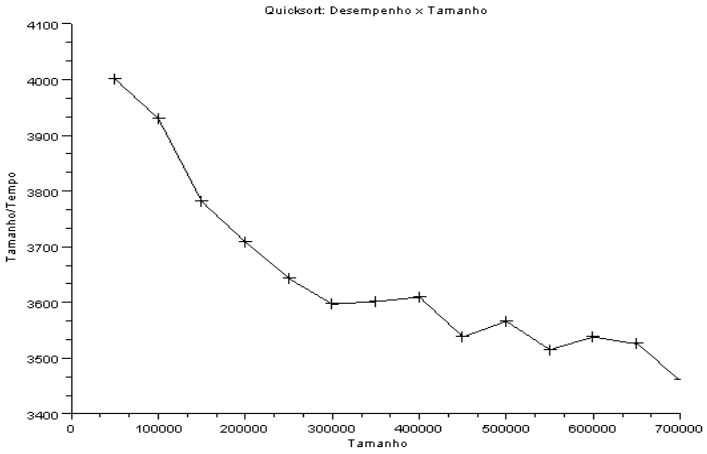


Figura 368: Elementos ordenados por unidade de tempo, em função do tamanho do vetor

10.4 Dividir para Conquistar

Os algoritmos de ordenação por intercalação e por partição são exemplos de emprego de uma estratégia clássica em computação, chamada *dividir para conquistar*. Dado um problema de tamanho n,

- o problema é dividido em 2 ou mais subproblemas, cada um de tamanho estritamente menor que n;
- cada subproblema é "conquistado" por aplicação recursiva da mesma estratégia, a não ser que o seu tamanho seja suficientemente pequeno para permitir uma solução direta;
- as soluções dos subproblemas são combinadas para resolver o problema original.

Na ordenação por intercalação a divisão em subproblemas é simples, mas a combinação das soluções dos subproblemas exige a intercalação. Na ordenação por partição a divisão em subproblemas exige a execução do algoritmo de partição, mas a combinação das soluções dos subproblemas consiste apenas em sua justaposição.

10.5 Ordenação por Radical

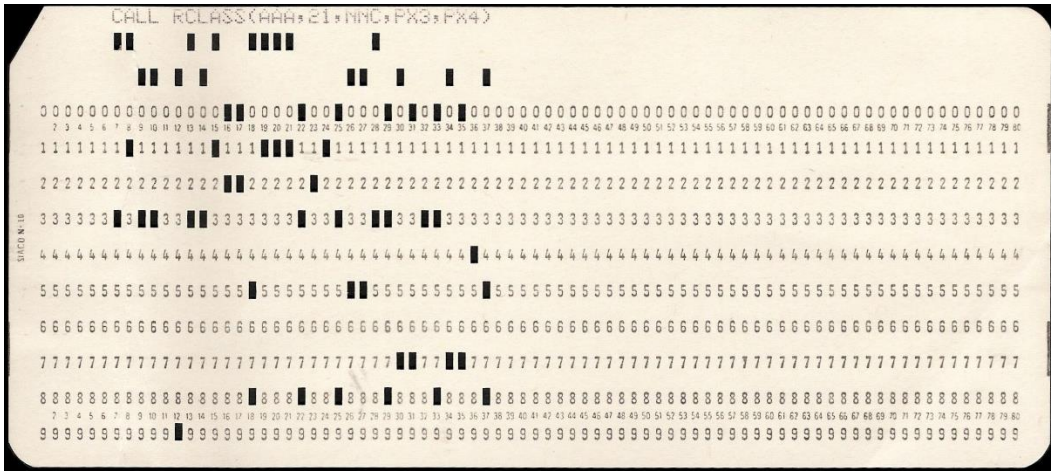


Figura 369: Um cartão perfurado[51]

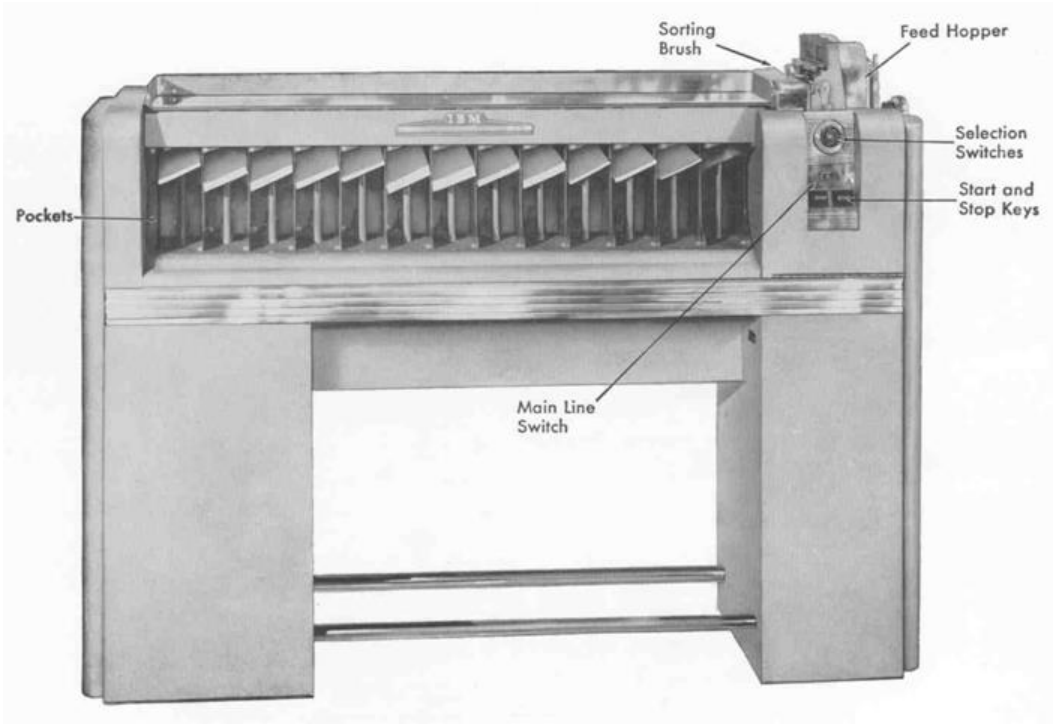


Figura 370: A classificadora de cartões IBM Type 82, de 1949[52]

Todos os algoritmos de ordenação que vimos na Seção 10 são algoritmos de ordenação por comparação, isto é, algoritmos que usam comparações entre os elementos a serem ordenados para efetuar seu trabalho. Em algumas situações é possível obter algoritmos de outro tipo, utilizando características particulares dos elementos a serem ordenados. Por exemplo, se os elementos a serem ordenados são todos números decimais com k algarismos, o algoritmo de ordenação por radical (*radix sort*, em inglês) pode ser utilizado.

Para explicar o algoritmo de ordenação por radical o melhor é mostrar como uma classificadora de cartões perfurados (veja a foto na Figura 370) era utilizada para ordenar um conjunto de cartões segundo valores presentes em, digamos, três colunas dos cartões. Uma pilha de cartões não ordenados era colocada no alimentador da classificadora, cujos controles

[51] "Punched_card.jpg (imagem JPEG, 2406×1160 pixels) - Redimensionada (29%)", acessado 22 de junho de 2013, https://upload.wikimedia.org/wikipedia/commons/f/f3/Punched_card.jpg.
[52] "IBM Card Sorters", acessado 22 de fevereiro de 2012, http://www.columbia.edu/cu/computinghistory/sorter.html#references.

eram ativados para distribuir os cartões nos escaninhos segundo o valor perfurado na coluna menos significativa.

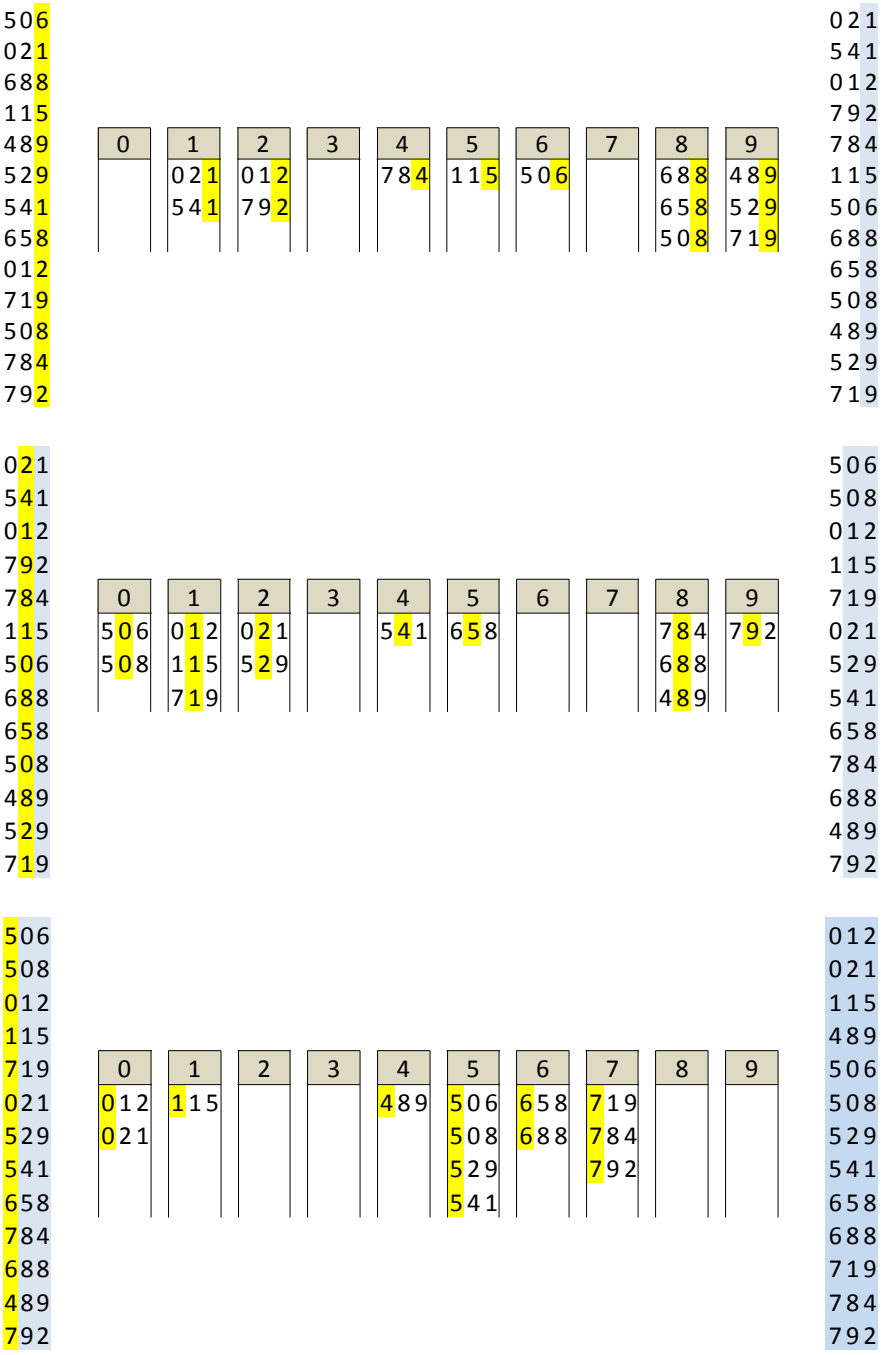


Figura 371: Etapas para ordenação por radical

Os cartões eram então recolhidos para formar uma nova pilha, respeitando a sequência dos escaninhos, de tal forma que os cartões na nova pilha estavam ordenados pelo algarismo menos significativo. O mesmo procedimento é realizado regulando a classificadora para distribuir pelo segundo algarismo menos significativo, depois com o terceiro menos significativo, até se atingir o k-ésimo algarismo, quando o vetor estará ordenado. A Figura 371 mostra as três distribuições necessárias para a ordenação completa de um conjunto de "cartões".

11 Algoritmos Numéricos

Nesta seção nós apresentamos alguns algoritmos numéricos para o cálculo de integrais definidas e também para encontrar raízes de uma função $f(x)$. Tais algoritmos têm utilidade para problemas cuja solução analítica é difícilmente obtida ou não existe. Nós desenvolvemos também uma função para o cálculo de e^x utilizando série de Taylor, cálculo que está sujeito a problemas que podem resultar de operações de truncamento e de arredondamento que decorrem do uso de um número fixo de bits na representação de ponto flutuante. Mais uma vez, estaremos desenvolvendo funções que já existem no Scilab: `integrate` e `intg` calculam integrais definidas, `fsolve` resolve sistemas de equações não lineares, e `exp` calcula e^x. Os métodos numéricos utilizados nas funções Scilab são muito mais sofisticados do que os que apresentamos aqui.

11.1 Integração por Trapézios

O primeiro passo para se obter uma aproximação numérica de uma integral $\int_a^b f(x)dx$ é a divisão do intervalo $[a, b]$ em n subintervalos iguais.Com isso nós vamos obter $n + 1$ pontos regularmente espaçados, que vamos chamar de $x_1, x_2, \cdots, x_{n+1}$. Nós temos $x_1 = a$, $x_{n+1} = b$ e $x_{i+1} - x_i = \Delta_x = (b - a)/n$, para todo $1 \leq i \leq n$.

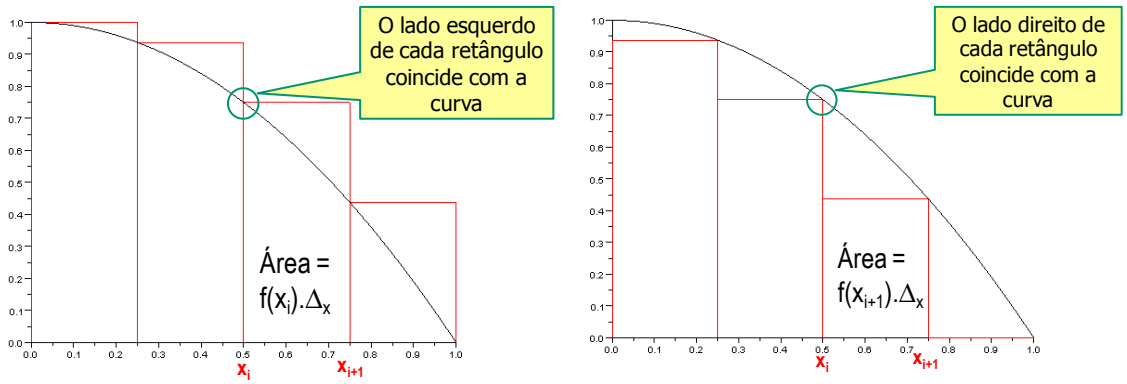


Figura 372: Áreas cobertas pelas somas de Riemann pela esquerda e pela direita para $f(x) = 1 - x^2$ no intervalo [0,1]

A Figura 372 e a Figura 373mostram duas formas de se aproximar do valor da integral utilizando somas das áreas de retângulos, conhecidas como *soma de Riemann* pela esquerda e soma de Riemann pela direita.

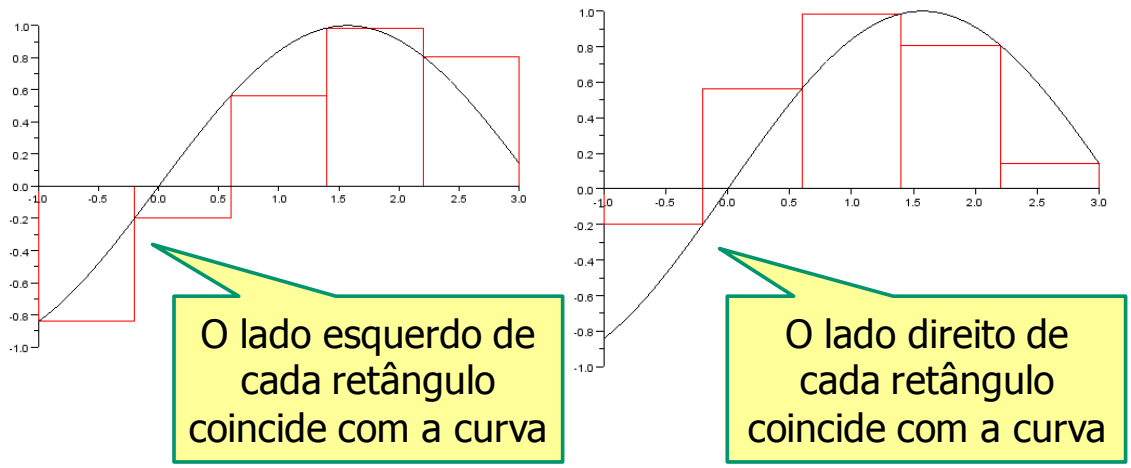


Figura 373: Somas de Riemman pela esquerda e pela direita para a função seno no intervalo [-1 3]

Podemos ver que a área definida pela soma de Riemann pela esquerda é dada por

$$A_{esq} = f(x_1).\Delta_x + f(x_2).\Delta_x + \cdots + f(x_n).\Delta_x = \Delta_x \sum_{i=1}^{n} f(x_i)$$

enquanto a área definida pela soma de Riemann pela direita é dada por

$$A_{dir} = f(x_2)\Delta_x + f(x_3)\Delta_x + \cdots + f(x_{n+1})\Delta_x = \Delta_x \sum_{i=2}^{n+1} f(x_i)$$

A medida em que o número de intervalos aumenta e o tamanho do intervalo diminui, as somas de Riemann vão se aproximando da área sob a curva, como mostra a Figura 374.

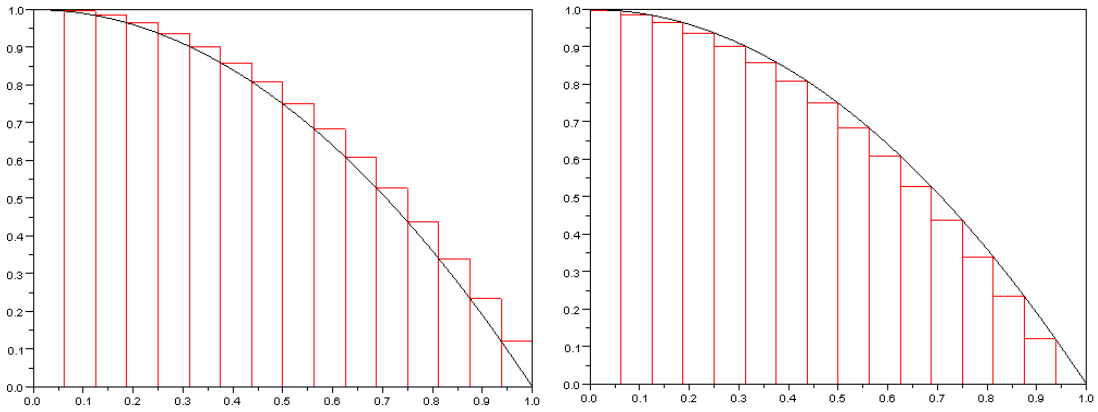


Figura 374: Somas de Riemann com 16 intervalos

As fórmulas das somas de Riemann levam diretamente às implementações em Scilab, mostradas na Figura 375 e na Figura 376.

```
function lrs = LeftRiemannSum(f,a,b,n)
//  Calcula a soma de Riemann esquerda da função
//  f entre os pontos a e b com n intervalos
  x = linspace(a,b,n+1);
  delta_x = (b-a)/n;
  lrs = sum(f(x(1:n))) * delta_x;
  // Comente os comandos abaixo se você não
  // quiser o gráfico
  scf();
  plot2d2(x,f(x),style=5,axesflag=5);
  plot2d3(x,f(x),style=5,axesflag=5);
  xx=linspace(a,b,50);
  plot2d(xx,f(xx));
endfunction
```

Figura 375: Função para o cálculo da soma de Riemann esquerda

```
function rrs = RightRiemannSum(f,a,b,n)
//  Calcula a soma de Riemann direita da função
//  f entre os pontos a e b com n intervalos
  x = linspace(a,b,n+1);
  delta_x = (b-a)/n;
  rrs = sum(f(x(2:n+1)))* delta_x);
  // Comente os comandos abaixo se você não
  // quiser o gráfico
  scf();
  plot2d2(x,[f(x(2:n+1)) f(b)],style=5,axesflag=5);
  plot2d3(x,[f(x(2:n+1)) f(b)],style=5,axesflag=5);
  xx=linspace(a,b,50);
  plot2d(xx,f(xx),axesflag=5);
endfunction
```

Figura 376: A função RightRiemannSum

Nestas funções, você pode observar que:

- O primeiro parâmetro de entrada, `f`, é a função cuja soma de Riemann será calculada;
- Os parâmetros `a` e `b` são os extremos do intervalo, e `n` é o número de sub-intervalos;
- A função `sum` do Scilab é utilizada para o cálculo da soma dos elementos de um vetor;
- Para o cálculo das somas somente os três primeiros comandos são necessários; os comandos restantes se destinam ao desenho de gráficos que ilustra a soma realizada.

Não é difícil ver que é possível obter uma aproximação melhor da área sob a curva em um sub-intervalo se utilizarmos o trapézio definido pelo valor da função nos limites de cada sub-intervalo, cono mostrado na Figura 377.

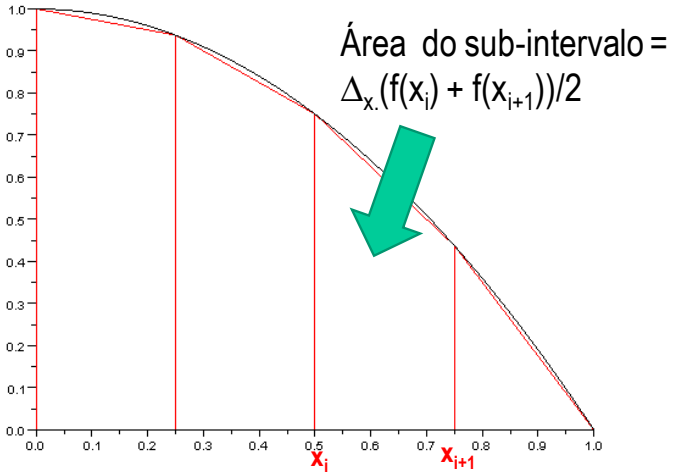


Figura 377: Aproximação por trapézios

A soma das áreas dos trapézios é dada por

$$\begin{aligned} A &= \frac{f(x_1)+f(x_2)}{2}\Delta_x + \frac{f(x_2)+f(x_3)}{2}\Delta_x + \cdots + \frac{f(x_{n-1})+f(x_n)}{2}\Delta_x + \frac{f(x_n)+f(x_{n+1})}{2}\Delta_x \\ &= \Delta_x\left[\frac{f(x_1)}{2} + f(x_2) + f(x_3) + \cdots + f(x_n) + \frac{f(x_{n+1})}{2}\right] \\ &= \Delta_x\left[\frac{f(x_1)+f(x_{n+1})}{2} + \sum_{i=2}^{n} f(x_i)\right] \end{aligned}$$

Uma função que calcula uma integral por trapézios está mostrada na Figura 378.

```
function area = TrapezoidalSum(f,a,b,n)
//  Calcula a área sob a curva f entre a e b,
//  utilizando n intervalos e a fórmula dos
//  trapézios
  x = linspace(a,b,n+1);
  delta_x = (b-a)/n;
  area = ( (f(x(1))+f(x(n+1)))/2 + ...
           sum(f(x(2:n))) ...
         )*delta_x;
// Os comandos abaixo desenham um gráfico
  scf();
  plot2d(x,f(x),style=5,axesflag=5)
  plot2d3(x,f(x),style=5,axesflag=5);
  xx=linspace(a,b,50);
  plot2d(xx,f(xx),axesflag=5);
endfunction
```

Figura 378: Função para aproximação de integrais por trapézios

Para testar esta função, vamos calcular $\int_0^{\pi} \sin(x)\,dx$, cujo valor exato nós sabemos:

$$\int_0^{\pi} \sin x\,dx = -\cos\pi - (-\cos 0) = 2$$

Com 5 intervalos, o resultado da chamada `TrapezoidalSum(sin,0,%pi,5)` foi 1.9337656; com 50 intervalos (chamada `TrapezoidalSum(sin,0,%pi,50)`), o valor obtido foi 1.999342.

11.2 Bisseção

Nós sabemos que as raízes de funções (ou *zeros de funções*) como um polinômio de 2º grau podem ser encontradas por fórmulas analíticas, mas isto não é possível para muitas outras funções. O método da bisseção é um algoritmo que serve para determinar numericamente uma raiz de uma equação $f(x) = 0$, onde $f(x)$ é uma função contínua qualquer. Para dar início ao algoritmo, precisamos de dois pontos a e b, sendo $a < b$, onde a função f assume sinais opostos, ou seja, ou bem $f(a) > 0$ e $f(b) < 0$, ou então $f(a) < 0$ e $f(b) > 0$, como mostrado na Figura 379.

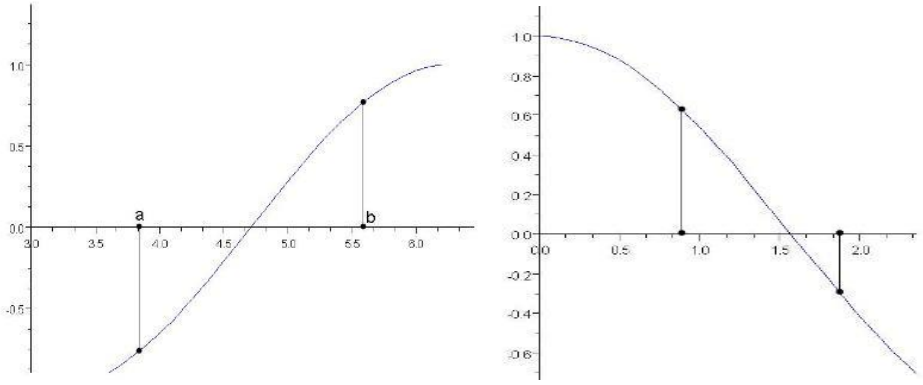


Figura 379: Exemplos de intervalos adequados para o método da bisseção

Como a função é contínua, e como $f(a)$ e $f(b)$ têm sinais opostos, em pelo menos um ponto r no intervalo $[a, b]$ a função será "obrigada" a cruzar o eixo dos x, onde $f(r) = 0$. É possível que o intervalo $[a, b]$ contenha mais de uma raiz, como na Figura 380.

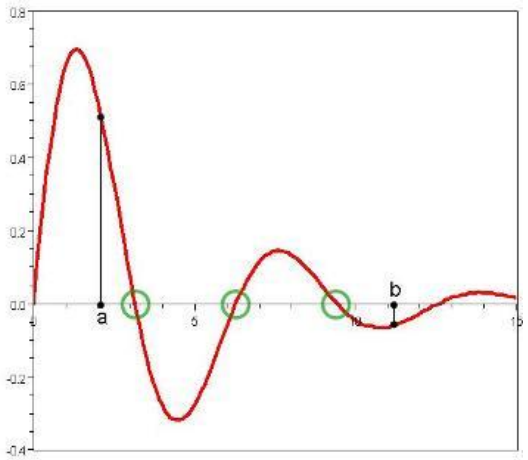


Figura 380: Intervalo contendo mais de uma raiz de uma função

Se $f(a)$ e $f(b)$ têm o mesmo sinal, o intervalo $[a, b]$ pode conter ou não uma raiz, como mostra a Figura 381.

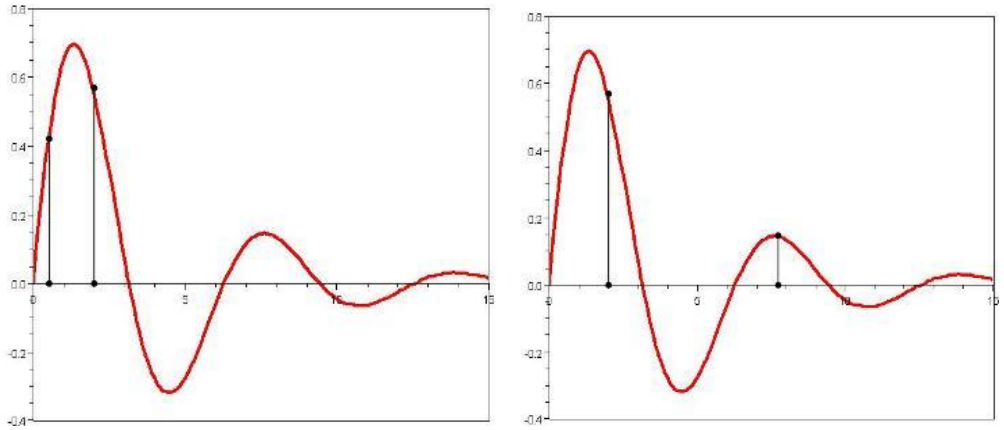


Figura 381: Intervalos em que a função não tem sinais opostos nos extremos podem conter ou não uma raiz

Se $f(x)$não for contínua, é possível que mesmo com $f(a).f(b) < 0$ (ou seja, $f(a)$ e $f(b)$ têm sinais opostos) não exista nenhuma raiz no intervalo $[a, b]$.

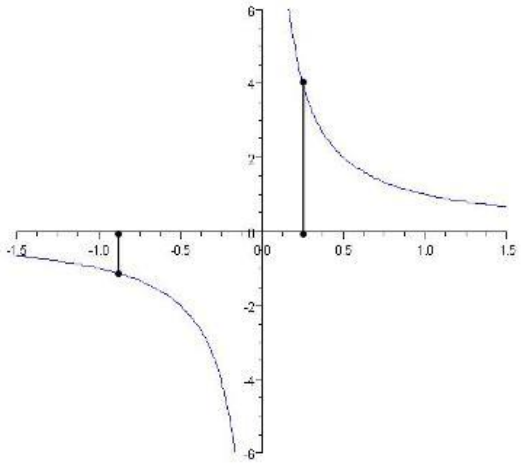


Figura 382: Se a função não for contínua, sinais opostos nas extremidades de um intervalo não garantem que ele contenha uma raiz

O método da bisseção exige portanto que $f(x)$ seja contínua em um intervalo $[a, b]$ tal que $f(a).f(b) < 0$. De uma forma similar ao algoritmo de pesquisa binária, a cada iteração a função f é calculada no ponto médio do intervalo, $m = (a + b)/2$.

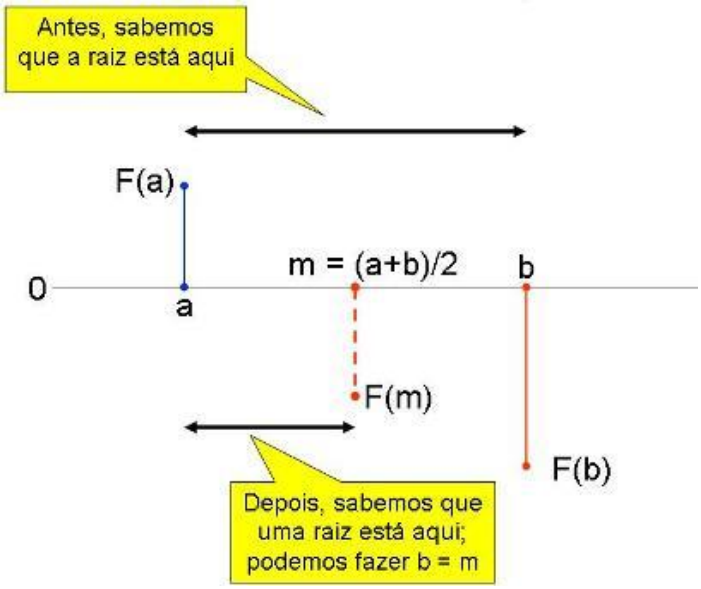


Figura 383: Caso em que a raiz está "à esquerda" do ponto médio do intervalo

Temos três casos possíveis. No primeiro (Figura 383) verificamos que $f(a).f(m) < 0$, e portanto que o intervalo $[a, m]$ contém pelo menos uma raiz, onde o algoritmo pode ser reaplicado.

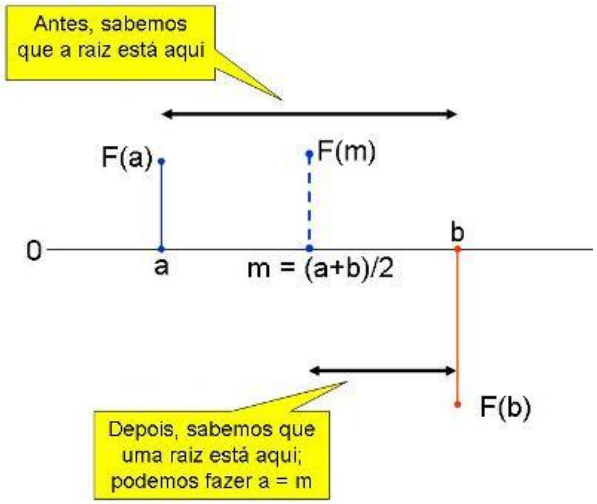


Figura 384: Caso em que a raiz está "à direita" do ponto médio do intervalo

No segundo caso (Figura 384) temos $f(m).f(b) < 0$, e é o intervalo $[m, b]$ que contém pelo menos uma raiz.

A terceira possibilidade é de termos tido sorte e encontrado m tal que $f(m) = 0$.

Ao fim de cada iteração, ou bem a raiz foi encontrada, ou o intervalo de busca foi reduzido à metade. O algoritmo consiste na aplicação repetida deste passo, e termina quando o intervalo onde se encontra a raiz é suficientemente pequeno para a precisão desejada.

Podemos agora dar início ao desenvolvimento de uma função que encontre uma raiz de uma outra função utilizando o método da bisseção.

```
function r = bissecao(f,a,b,tol)
//    se f é contínua e se f(a).f(b) < 0, esta
//    função calcula a raiz r com precisão menor ou igual
//    ao valor de tol
endfunction
```

Figura 385: Cabeçalho da função bissecao

A Figura 385 mostra uma primeira versão da função **bissecao** contendo apenas o seu cabeçalho:

- O primeiro parâmetro formal de entrada, **f**, é a função da qual se deseja encontrar a raiz;
- Os parâmetros **a** e **b** são os limites do intervalo, e devem ser tais que **f(a)*f(b) < 0**;
- O parâmetro **tol** é a tolerância, isto é, um valor para o tamanho do intervalo de busca onde a precisão desejada é considerada satisfatória;
- O parâmetro de saída **r** é a raiz encontrada.

Para testar a função **bissecao**, nós precisamos:

- de uma função contínua,
- de um intervalo onde a função troca de sinal,
- e de conhecer o valor de uma raiz nesse intervalo para que seja possível verificar o resultado.

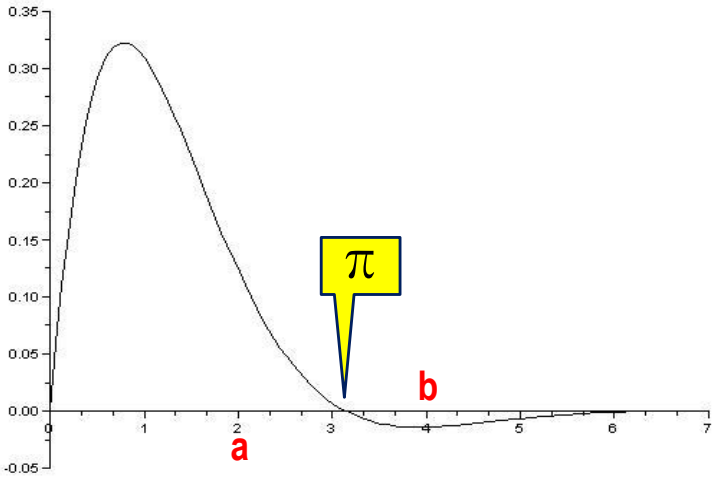


Figura 386: Gráfico de $f(x) = e^{-x} \sin x$

A função $f(x) = e^{-x} \sin x$, cujo gráfico está mostrado na Figura 386, satisfaz a estes requisitos, pois:

- é contínua;
- tem sinais opostos nos extremos do intervalo $[2,4]$, e portanto este intervalo contém uma raiz, e
- essa raiz é π.

A Figura 387 mostra uma implementação desta função. Repare no uso do operador **.***, de multiplicação elemento a elemento, ao invés do operador *, de multiplicação matricial.

```
function y = exp_sin(x)
  y = exp(-x) .* sin(x);
endfunction
```

Figura 387: A função exp_sin

O gráfico da Figura 386 pode ser obtido na console do Scilab com os comandos mostrados na Figura 388. A diretiva `axesflag=5` faz com que os eixos sejam traçados no ponto (0,0).

```
-->exec("exp_sin.sci")
-->x = linspace(0,2*%pi,101);
-->y = exp_sin(x);
-->plot2d(x,y,axesflag=5)
```

Figura 388: Comandos para obter o gráfico da Figura 386 na console do Scilab

Com isso nós já podemos construir um programa testador para a função `bissecao`, conforme mostra a Figura 389. O programa permite experimentar com diversos valores de tolerância, comparando a raiz calculada com o valor de π com 10 casas decimais.

```
clear
exec("exp_sin.sci");
exec("bissecao.sci");
tolerancia = input("\nTolerância = ");
while tolerancia > 0
  raiz = bissecao(exp_sin,2,4, tolerancia);
  printf(" Raiz = %12.10f; \n   Pi = %12.10f\n",raiz,%pi);
  tolerancia = input("\nTolerância = ");
end
```

Figura 389: O programa bissecao_teste.sce

Um primeiro refinamento da função `bissecao.sci` é o loop que, a cada passo, reduz à metade o intervalo que contém a raiz. No código mostrado na Figura 390 nós podemos observar que:

- o loop é interrompido quando o tamanho do intervalo é menor ou igual à tolerância fornecida pelo usuário, e
- o valor retornado como raiz é o ponto médio do intervalo.

```
function r = bissecao(f,a, b, tol)
//    se f é contínua e se f(a).f(b) < 0, esta
//    função calcula a raiz r com precisão menor ou igual
//    ao valor de tol
  while b-a > tol
    // Redução do intervalo que contém a raiz
  end
  r = (a+b)/2;
endfunction
```

Figura 390: Primeiro refinamento da função bissecao

O refinamento da redução do intervalo é simplesmente a codificação do método da bisseção, como mostra a Figura 391. Repare que:

- `quando f(a)*f(m) < 0`, a função faz `b = m`;
- quando `f(b)*f(m) < 0`, a função faz `a = m`;
- quando `f(m) == 0`, a função faz `a = m` e `b = m`.

```
function r = bissecao(f,a, b, tol)
//    se f é contínua e se f(a).f(b) < 0, esta
//    função calcula a raiz r com precisão menor ou igual
//    ao valor de tol
  while b-a > tol
    // Redução do intervalo que contém a raiz
    m = (a+b)/2; //Ponto médio
    if f(a)*f(m) <= 0 then
      // [a,m] contém uma raiz
      b = m;
    end
    if f(m)*f(b) <= 0 then
      // [m,b] contém uma raiz
      a = m;
    end
  end
  r = (a+b)/2;
endfunction
```

Figura 391: A função bissecao

A Figura 392 mostra o resultado de um teste da função `bissecao`, onde podemos ver o efeito da tolerância fornecida pelo usuário sobre a precisão do cálculo da raiz.

```
Tolerância = 1.0e-3
 Raiz = 3.1411132813;
   Pi = 3.1415926536

Tolerância = 1.0e-6
 Raiz = 3.1415925026;
   Pi = 3.1415926536

Tolerância = 1.0e-10
 Raiz = 3.1415926536;
   Pi = 3.1415926536
```

Figura 392: Teste da função bissecao

11.3 Série de Taylor para exp(x) e Cancelamento Catastrófico

Do cálculo sabe-se que, para qualquer x real, e^x pode ser calculado por uma série de Taylor, que é uma soma de infinitos termos com a forma abaixo:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

ou, lembrando que $0! = 1$,

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

Esta série converge para qualquer valor de x. Nós podemos ver que, a partir do termo onde $i \geq x$, $i!$ cresce mais rapidamente que x^i, e que o valor absoluto dos termos tende para zero quando i tende para infinito. Queremos aqui construir uma função Scilab que calcule e^x por esta fórmula, adicionando termos até que seu valor absoluto seja menor que uma tolerância fornecida pelo usuário. Já temos como escrever o cabeçalho da função, como mostra a Figura 393.

```
function y = expTaylor(x,tol)
// Calcula a soma dos termos
// da série de Taylor até o primeiro
// termo com valor absoluto menor
// que a tolerância tol
endfunction
```

Figura 393: Cabeçalho da função expTaylor

Para testar esta função, o programa **expTaylor_teste**, mostrado na Figura 389, lê um valor para a tolerância. Depois, repetidamente, lê valores para a variável **x**, calcula e compara os valores retornados pela função **expTaylor** que nós desenvolvemos e pela função **exp** fornecida pelo Scilab, que é bastante confiável pois utiliza técnicas muito sofisticadas de cálculo numérico.

```
exec("expTaylor.sci");
tol = input("\ntol = ");
x = input("\nx = ");
while x ~= 999
  expCalc = expTaylor(x,tol);
  printf("\n           x      exp(x)      expTaylor(x)      Erro")
  printf ("\n%12g %15.8e %15.8e %15.8e\n", ...
          x,exp(x),expCalc,exp(x)-expCalc)
  x = input("\nx = ");
end
```

Figura 394: O programa expTaylor_teste.sce

Para o desenvolvimento da função **expTaylor**, nós devemos reparar que é possível obter t_i, o i-ésimo termo da série a partir do termo anterior, pois

$$t_i = \frac{x^i}{i!} = \frac{x^{i-1}}{(i-1)!}\frac{x}{i} = \frac{x}{i}t_{i-1}$$

Com isso nós chegamos à forma final da função **expTaylor**, mostrada na Figura 395.

```
function y = expTaylor(x,tol)
// Calcula a soma dos termos
// da série de Taylor até o primeiro
// termo com valor absoluto menor
// que a tolerância tol
  Termo = 1;
  y = 1;
  i = 1;
  while abs(Termo) >= tol
    Termo = Termo * x / i;
    y = y + Termo;
    i = i+1;
  end
endfunction
```

Figura 395: A função expTaylor

Vamos primeiramente testar a função para alguns valores positivos de **x**. Podemos ver na Figura 396 que os resultados são muito bons, com diferenças 16 ordens de grandeza menores que os valores calculados pelas duas funções.

```
tol = 1.0e-40
x = 1
              1 2.71828183e+000 2.71828183e+000 -4.44089210e-
016
x = 10
             10 2.20264658e+004 2.20264658e+004 7.27595761e-
012
x = 30
             30 1.06864746e+013 1.06864746e+013 -3.90625000e-
003
```

Figura 396: Resultados de testes da função expTaylor com x positivo

Mas o teste com valores negativos nos reserva surpresas desagradáveis, como mostra a Figura 397. Para `x == -1`, o erro é inferior aos valores por 15 ordens de grandeza, muito bom. Para `x == -10`, o erro é 8 ordens de grandeza menor que os valores calculados; vá lá. Já com `x == -20`, o erro é da mesma ordem de grandeza dos valores calculados, muito ruim. A casa cai mesmo com `x==-30`, quando o erro é 9 ordens de grandeza maior que o valor correto, e, pior, o valor calculado para e^{-30} é negativo, sendo que e^x é uma função estritamente positiva!

```
tol = 1.0e-40
x = -1
             -1 3.67879441e-001 3.67879441e-001 -1.11022302e-
016
x = -10
            -10 4.53999298e-005 4.53999296e-005 1.39453573e-
013
x = -20
            -20 2.06115362e-009 5.62188447e-009 -3.56073085e-
009
x = -30
            -30 9.35762297e-014 -3.06681236e-005 3.06681237e-
005
```

Figura 397: Resultados de testes da função expTaylor com x negativo

O que aconteceu? A fórmula para a série de Taylor é provada matematicamente, e a função expTaylor é uma implantação direta da fórmula, com pouca possibilidade de erros.

```
-->eps = 1.0e-23;
-->y = 1.0e23;
-->x = y + eps;
-->x == y
 ans  =
  T
```

x é igual a y bit por bit!

Figura 398: Exemplo de cancelamento catastrófico

A origem dos maus resultados está na aritmética de ponto flutuante, que usa um número fixo de bits para representação da mantissa. Operações aritméticas com números com grandes diferenças de ordem de grandeza não funcionam corretamente, como mostra a Figura 398.

O valor `1.0e-23` somado a `1.0e23` não altera o seu expoente, o que é natural, mas tampouco altera a sua mantissa, que não possui bits suficientes para essa adição. O valor somado é simplesmente perdido na operação, em um efeito que é conhecido por *cancelamento catastrófico*.

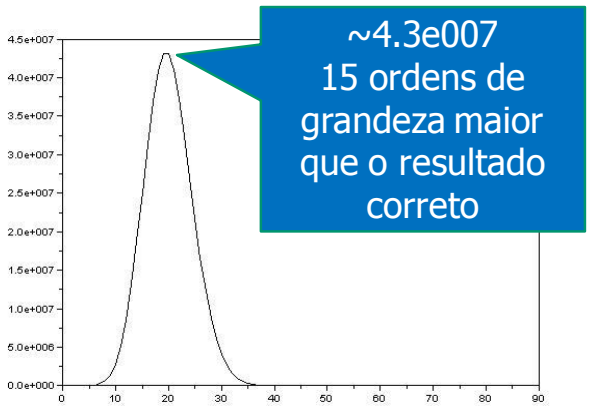


Figura 399: Gráfico com valores absolutos dos termos da série de Taylor para x == -20

A Figura 399 mostra um gráfico com os valores absolutos dos termos da série de Taylor para x `== -20`. O maior valor absoluto de um termo é da ordem de 10^7, e o valor correto para e^{-20} é da ordem de 10^{-9}. Ou seja, nas operações aritméticas realizadas com os primeiros termos da série os erros de truncamento podem ser bem maiores que o resultado final, e é isso o que ocorre com o uso da função `expTaylor` para o cálculo de exponenciais de números negativos.

Como lição a tirar destes exemplos, você deve ter muito cuidado ao operar com números de valores com grandes diferenças de ordem de grandeza. A aritmética de ponto flutuante é melindrosa; use funções de bibliotecas, desenvolvidas por profissionais de cálculo numérico, sempre que possível. Por outro lado, não se deixe levar pelo pessimismo. Programas numéricos funcionam como esperado na maior parte dos casos.

12 Complexidade de Problemas

Para o problema da fatoração vimos um primeiro algoritmo com complexidade $O(2^n)$, e um outro, superior, com complexidade $O(2^{n/2})$. Para o problema da ordenação, vimos o algoritmo de Seleção, com complexidade $O(n^2)$, e o MergeSort, com complexidade $O(n \log n)$, bem superior. Criatividade e estudo de propriedades de um problema podem levar a melhores algoritmos; mas isto seria sempre possível? Esta é a questão central deste módulo, que leva a conceitos fundamentais da Ciência da Computação.

12.1 Complexidade da Ordenação

É fácil ver que a complexidade do algoritmo de ordenação por radical é $O(n)$, superior à complexidade no pior caso do melhor algoritmo de ordenação por comparação que vimos, o MergeSort, que é $O(n \log n)$. Mas o algoritmo de ordenação por radical é menos genérico. Se considerarmos apenas algoritmos de ordenação que só utilizam comparações a situação é diferente. Nós vamos agora ao objetivo central desta seção, que é demonstrar que não existe nenhum algoritmo de ordenação por comparações que tenha complexidade melhor que $O(n \log n)$.

Um algoritmo de ordenação por comparações pode receber como entrada um vetor com elementos dispostos em qualquer ordem, e produz uma permutação destes elementos disposta em ordem crescente. Durante a sua execução o algoritmo faz diversas comparações entre os elementos do vetor de entrada.

Nós vamos aqui modificar ligeiramente alguns algoritmos de ordenação, fazendo com que eles mantenham durante toda a sua execução um registro das posições que cada elemento ocupava no vetor original. Essencialmente essa modificação consiste em adicionar um vetor de índices com valores iniciais [1 2 3 ... n] e, a cada troca efetuada com o vetor a ser ordenado, trocar também os elementos correspondentes no vetor de índices. Assim, como a ordenação de [321 12 609] produz o vetor [12 321 609], o vetor de índices ao final da execução do algoritmo deverá, neste caso, conter [2 1 3], significando que o primeiro elemento no vetor ordenado tinha a posição 2 no vetor original, o segundo elemento no vetor ordenado tinha a posição 1 no vetor original, e que o terceiro elemento no vetor ordenado tinha a posição 3 no vetor original. Nós vamos também fazer com que os algoritmos de ordenação registrem as posições no vetor original dos elementos envolvidos em cada comparação feita durante sua execução.

Todo algoritmo de ordenação por comparações faz alterações em suas variáveis, depois uma comparação entre elementos do vetor a ser ordenado, alguma ação seja sobre este vetor (como uma troca de posições entre dois elementos), seja sobre outras das variáveis que utiliza; depois, outra comparação, e outra ação, até que o algoritmo termine. A sequência de comparações e de ações depende dos resultados das comparações efetuadas.

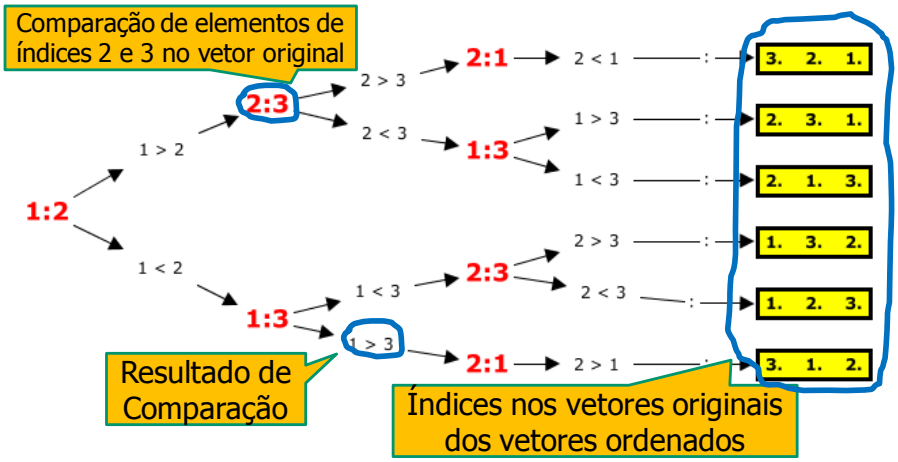


Figura 400: Árvore de decisões para o SelectSort

Uma *árvore de decisões* é uma forma compacta de descrever o comportamento de um algoritmo de ordenação por comparações quando submetido a diversos vetores de entrada. A Figura 400 mostra uma árvore de decisões gerada pelo SelectSort, quando submetido a todas as permutações de um vetor [a b c], onde a, b e c são valores distintos. A árvore de decisões deve ser "lida" da seguinte forma:

- a primeira comparação é sempre feita entre os elementos que estão nas posições iniciais 1 e 2;
- a segunda comparação depende do resultado da primeira:
  - se o elemento de posição inicial 1 for maior que o elemento de posição inicial 2, a segunda comparação é feita entre os elementos de posições iniciais 2 e 3;
  - se o elemento de posição inicial 1 for menor que o elemento de posição inicial 2, a segunda comparação é feita entre os elementos de posições iniciais 1 e 3

E assim por diante. Toda rota na árvore de decisões termina em uma permutação com as posições no vetor inicial dos elementos do vetor ordenado. Nós vemos que o algoritmo de Seleção e Troca faz no sempre 3 comparações para ordenar um vetor de 3 elementos.

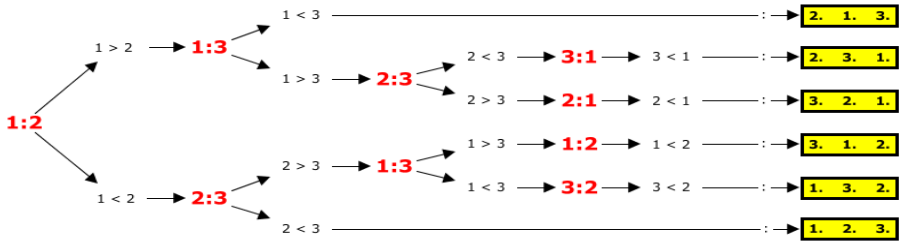


Figura 401: Árvore de decisões para o algoritmo da bolha

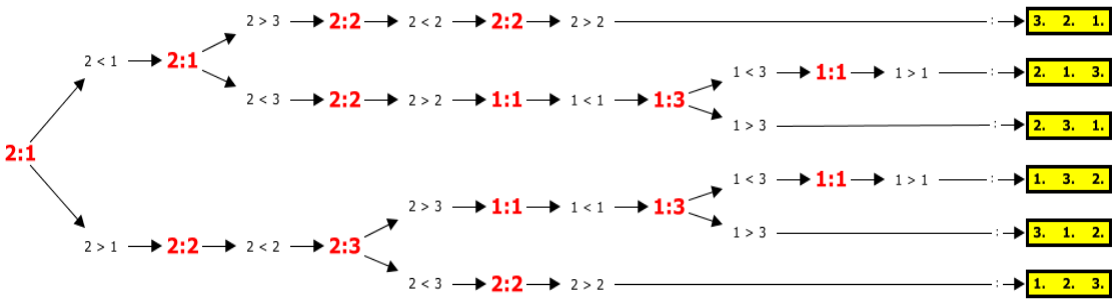


Figura 402: Árvore de decisões para o Quicksort

A Figura 401 mostra uma árvore de decisões para o algoritmo da Bolha, e a Figura 402 para o Quicksort. Nós podemos ver que o algoritmo da Bolha faz no mínimo 2 e no máximo 4 comparações para ordenar 3 elementos, e que o Quicksort faz no mínimo 4, e no máximo 6 comparações para ordenar os mesmos 3 elementos (isso mesmo, o Quicksort tem um comportamento sofrível para a ordenação de vetores pequenos).

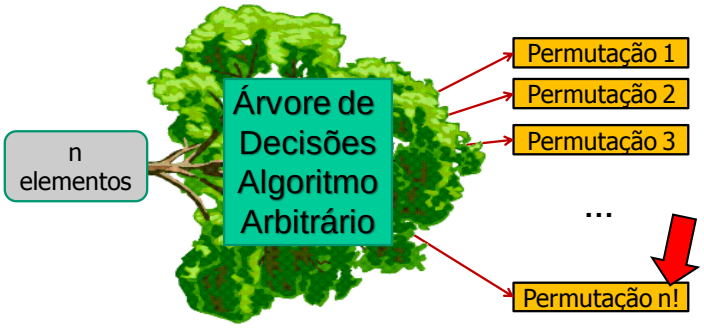


Figura 403: Árvore de decisões para um algoritmo arbitrário de ordenação por comparações

O termo *árvore* é usado em Ciência da Computação para designar estruturas hierárquicas. Árvores são compostas por nós; o nó no ponto mais alto da hierarquia é a *raiz da árvore*. Parentescos são usados para designar relações definidas por uma árvore: os nós imediatamente abaixo de um dado nó são seus filhos, o nó imediatamente acima de um nó é seu pai, e assim por diante. Nós sem filhos são chamados *folhas*. Árvores são comumente desenhadas de cabeça para baixo, com a raiz encima e as folhas embaixo. A Figura 404 mostra uma *árvore binária*, isto é, uma árvore onde cada nó tem no máximo 2 filhos.

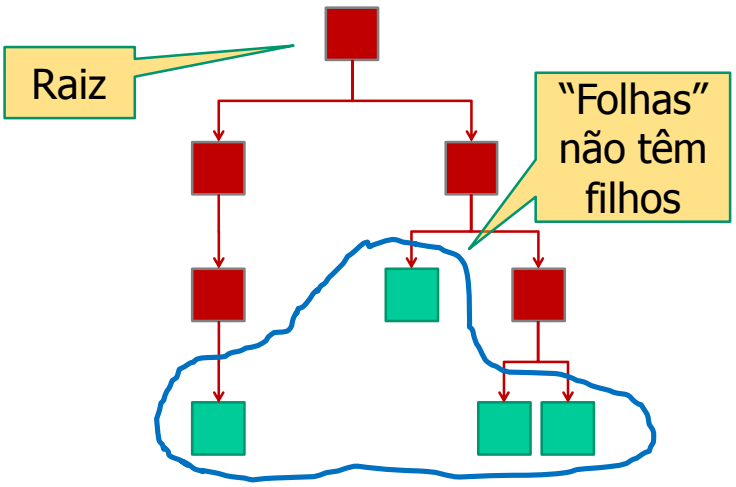


Figura 404: Uma árvore binária

A *profundidade* de um nó em uma árvore é o número de passos necessários para se chegar a ele, partindo da raiz (que, consequentemente, tem profundidade 0). A Figura 405 mostra as profundidades dos nós da árvore da Figura 404.

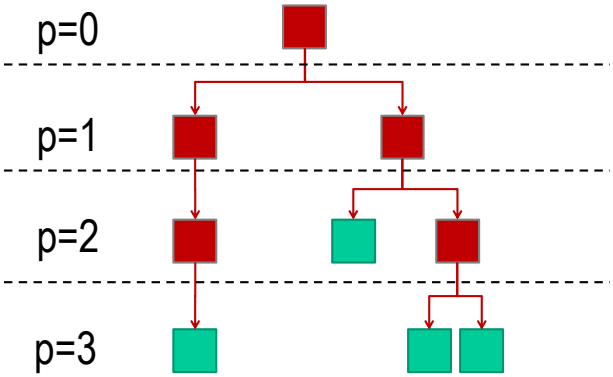


Figura 405: Profundidade em uma árvore

Uma árvore de decisões associada a um algoritmo de ordenação por comparação é claramente uma árvore binária, onde os nós são as comparações, e folhas são as comparações que levam ao término do algoritmo e portanto a uma permutação de índices. O número de comparações realizadas por um algoritmo é, no pior caso, a maior profundidade dessas folhas.

Qualquer algoritmo correto de ordenação por comparação deve ser capaz de ordenar qualquer permutação de um vetor com n elementos distintos. Cada permutação da entrada deve produzir, além do vetor ordenado, uma permutação dos índices no vetor inicial de saída. Portanto, para uma entrada de tamanho n, a árvore de decisões deverá ter $n!$ folhas, número total de permutações dos n elementos. Para conseguir um limite mínimo do número de comparações necessário para ordenar n elementos, válido para qualquer algoritmo de ordenação por comparações, precisamos estabelecer uma relação entre o número de folhas e a profundidade máxima de uma árvore binária.

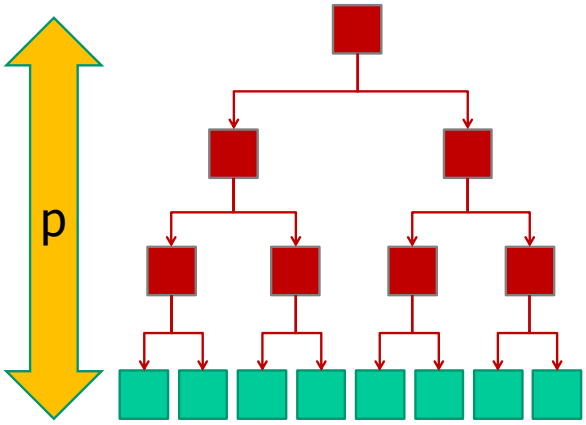


Figura 406: Uma árvore binária de profundidade p tem no máximo2^p folhas

Não é difícil acreditar que uma árvore binária de profundidade p tem no máximo 2^p elementos, fato ilustrado pela Figura 406. A menor profundidade máxima de uma árvore de decisões associada a um algoritmo de ordenação é dada portanto por:

$$2^p \geq n!$$

ou

$$p \geq log_2(n!) = log_2(1) + log_2(2) + \cdots + log_2(n)$$

Mas $\sum_{i=1}^{n} log_2(i)$ é uma aproximação – a soma de Riemann pela esquerda, com intervalos iguais a 1 – para a área sob a curva da função $log_2(x)$ entre $x = 1$ e $x = n$, como mostra a Figura 407.

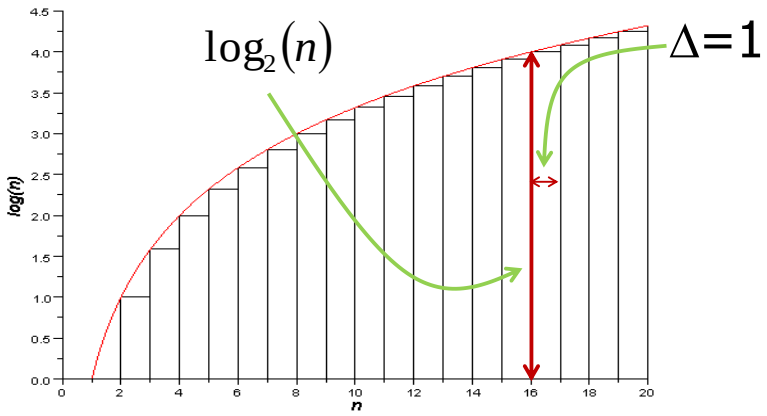


Figura 407: A soma dos logaritmos dos inteiros de 1 a n é a soma de Riemann pela esquerda para log(n)

Podemos então escrever

$$p \geq \sum_{i=1}^{n} log_2(i) \cong \int_1^n log_2(x)dx = (xlog_2(x) - x)\,|_1^n = nlog_2(n) - n + 1$$

Isso prova que o limite inferior para a complexidade de qualquer algoritmo de ordenação por comparações é $O(nlog_2 n)$.

12.2 Problemas NP-completos: O Problema do Caixeiro Viajante

Um caixeiro viajante precisa visitar n cidades, percorrendo a menor distância possível, sem passar duas vezes pela mesma cidade, e retornando à sua cidade de origem. Ele conhece a distância entre duas cidades quaisquer de seu roteiro; um exemplo está mostrado na Figura 408. Não existem estradas entre as cidades sem ligação no grafo.

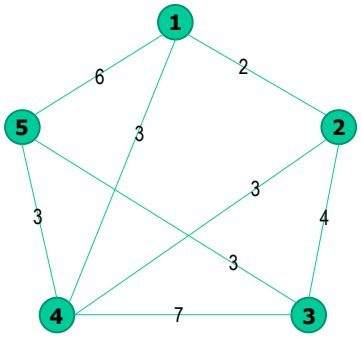
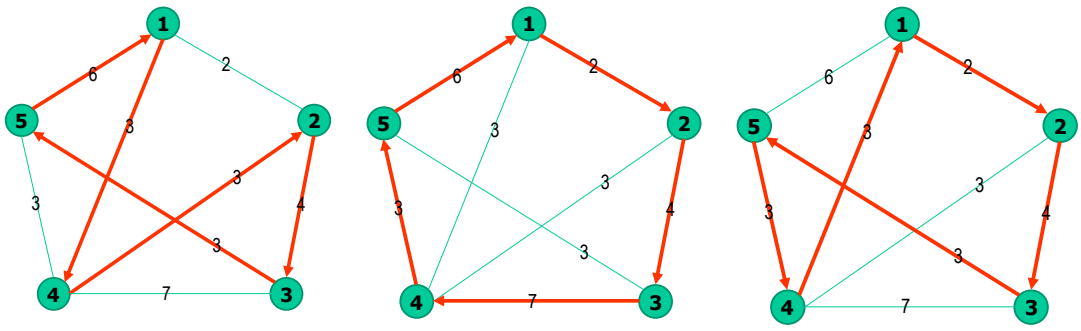


Figura 408: Distâncias entre cidades a serem visitadas pelo caixeiro viajante

Qual é o melhor roteiro, isto é, qual é o roteiro com a menor distância total, partindo da cidade 1? Na Figura 409 você pode ver como a escolha de um roteiro influencia a distância total.



D = 19 D = 22 D = 15

Figura 409: Algumas alternativas de roteiros para o caixeiro viajante

Um exame atento nos faz perceber que o roteiro mais à direita na Figura 409 é, dentre todos, o de menor custo. Descobrir o roteiro ótimo fica bem mais complicado quando temos mais cidades, como na Figura 410.

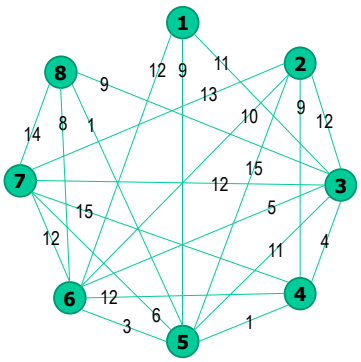
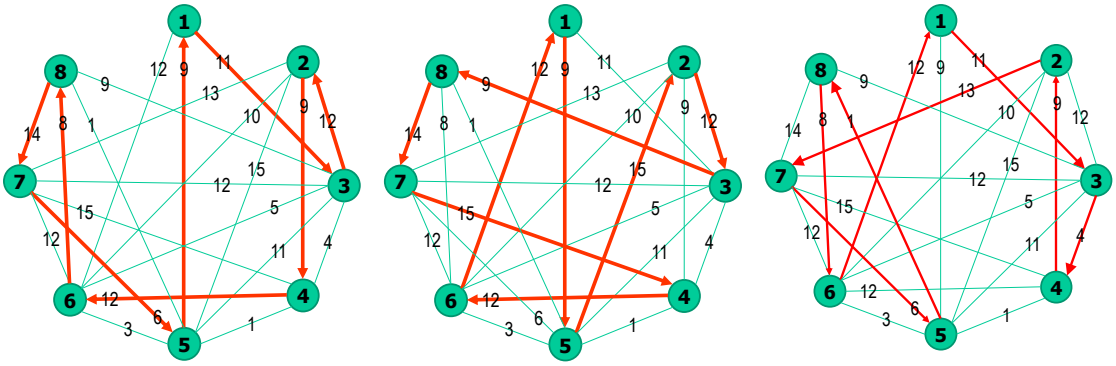


Figura 410: Um problema com 8 cidades

Temos muito mais alternativas a examinar; três exemplos estão na Figura 411. Aqui também a alternativa mais à direita é a rota ótima, fato que não é fácil de se confirmar por um simples exame do problema. Precisamos do auxílio de um computador.

Queremos construir um programa que, dado um conjunto de cidades e suas distâncias, descubra o melhor roteiro para o caixeiro viajante.



D = 81 D = 98 D = 64

Figura 411: Algumas alternativas de roteiros para o problema da Figura 410

Vamos tentar uma solução direta. Devemos:

1. Gerar todas as alternativas de roteiros;
2. Para cada roteiro, calcular o seu custo total;
3. Escolher o roteiro com menor distância total.

E como fazer para gerar todas as alternativas de roteiros? Temos que gerar todas as permutações das cidades a serem percorridas. Para um problema com 4 cidades, as rotas a serem examinadas são (lembrando que a cidade 1 é o ponto inicial e final, e portanto faz parte de qualquer circuito):

2 3 4

2 4 3

3 2 4

3 4 2

4 2 3

4 3 2

Nós iremos precisar de uma função que gere as permutações de um conjunto de números, cada número correspondendo a uma cidade. O Scilab já oferece a função **perms** para isso, mas vamos aqui desenvolver uma função equivalente. Vamos primeiramente definir um cabeçalho para uma função **permutations** (Figura 412).

```
Function p = permutations(A)
  // gera uma matriz p onde
  // cada linha é uma permutação de A
Endfunction
```

Figura 412: Cabeçalho da função permutations

Precisamos agora de um algoritmo para gerar essas permutações. O raciocínio é, mais uma vez, recursivo:

- Se o vetor **A** é de tamanho 1, ele já é a única permutação possível;
- senão, devemos "separar" o primeiro elemento do vetor;
- gerar todas as permutações dos elementos restantes (mesmo problema inicial, mas com um elemento a menos), e fazer uma justaposição do elemento separado com as permutações obtidas;
- repetir este procedimento para os demais elementos do vetor.

Como exemplo, considere o processo de gerar todas as permutações do vetor [2 3 4]. Nós devemos:

- Separar o elemento 2, e gerar todas as permutações do vetor [3 4]; depois, concatenar 2 às permutações obtidas;
- Separar o elemento 3, e gerar todas as permutações do vetor [2 4]; depois, concatenar 3 às permutações obtidas;
- Finalmente, separar o elemento 4, e gerar todas as permutações de [2 3], e depois concatenar 4 às permutações obtidas.

A função **permutations** mostrada na Figura 413 é uma implementação direta deste algoritmo.

```
function p = permutations(A)
  if length(A) == 1 then
    p = A;
  else
    p = [];
    for i = 1:length(A)
      B = permutations(OneOut(A,i));
      [nl,nc] = size(B);
      for j = 1:nl
        p = [p ; [A(i) B(j,:)]];
      end
    end
  end
endfunction
```

```
function b = OneOut(A,i)
  x = 1:length(A);
  b = A(x ~= i);
endfunction
```

Figura 413: A função permutations

Ela faz uso da função `OneOut`, que implementa a operação de "separar" o i-ésimo elemento do vetor, e que é um bom exemplo de uso das possibilidades oferecidas pelo Scilab para manipulação de matrizes apresentadas na Seção 6.7 (pág. 161).

Já podemos dar início ao desenvolvimento do programa principal. A Figura 414 apresenta uma primeira versão, formada apenas por comentários.

```
// Lê a matriz de distâncias
// Gera todas as rotas possíveis
// Calcula o custo de cada rota
// Seleciona a de menor custo
// Imprime o resultado
```

Figura 414: Primeira versão do programa CaixeiroViajante.sce

Vamos atacar inicialmente a leitura da matriz de distâncias. Queremos utilizar a função `fscanfMat` para esta leitura, mas temos que levar em conta que:

- precisamos representar o valor infinito para distâncias entre cidades sem conexão direta, e
- a função `fscanfMat` só lê números.

Isto pode ser resolvido adotando a convenção de usar no arquivo de entrada o valor -1 para representar infinito. Com isto podemos construir um arquivo de distâncias como mostra a Figura 415.

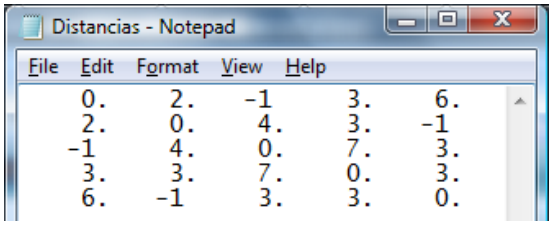


Figura 415 : Arquivo Distancias.txt com distâncias entre as cidades mostradas na Figura 408

Este arquivo pode ser lido diretamente com `fscanfMat`. Após a leitura, os elementos com valor `-1` devem ser substituídos por `%inf`. Você pode ver este código na Figura 416, onde está destacado o comando que faz as substituições.

```
// Lê a matriz de distâncias
Dist = ...
fscanfMat(uigetfile("*.txt",pwd(),"Distâncias"));
// Substitui -1 por %inf
Dist(Dist==-1) = %inf;
```

Figura 416: Leitura do arquivo com distâncias

Para gerar as rotas e calcular seus custos, vamos usar a função `permutations` da Figura 413, e também uma função `cost`, que recebe como parâmetros de entrada uma matriz `D` de distâncias, e um vetor `path`, que contém os índices das cidades que compõem uma rota.

```
[nl,nc] = size(Dist); //nl deve ser igual a nc
Rotas = permutations(2:nc);
[NL,NC] = size(Rotas);
for i = 1:NL
  Custo(i) = cost(Dist,[1 Rotas(i,:) 1]);
End
```

```
function c = cost(D,path)
  c = 0;
  for i=1:length(path)-1
    c = c + D(path(i),path(i+1));
  end
endfunction
```

Figura 417: Obtenção de todas as rotas e cálculo dos custos

Nos trechos de código mostrados na Figura 417 você deve reparar que:

- o vetor `Rotas` recebe todas as permutações das cidades de `2` a `n`, e não de `1` a `n`, pois a cidade `1` é sempre o ponto de partida e de chegada.
- a rota enviada (passada como parâmetro real) para a função `cost` é o vetor formado pela cidade `1`, acrescida das cidades que compõem uma linha do vetor `Rotas`, acrescido novamente pela cidade `1`.

```
// Resolve o problema do caixeiro viajante
clear()

exec("permutations.sci");
exec("cost.sci");
exec("OneOut.sci");
exec("SelecionaMenor.sci");
exec("PrintMatrix.sci");

// Lê a matriz de distâncias
Dist = fscanfMat(uigetfile("*.txt",pwd(),"Distâncias"));
PrintMatrix("Distâncias",Dist);
// Substitui -1 por %inf
Dist(Dist==-1) = %inf;
// Obtenção das rotas
[nl,nc] = size(Dist); //nl deve ser igual a nc
Rotas = permutations(2:nc);
// Calcula o custo de cada rota
[NL,NC] = size(Rotas);
for i = 1:NL
  Custo(i) = cost(Dist,[1 Rotas(i,:) 1]);
  if Custo(i) <> Exemplos(:,1) then
  end
end
// Seleciona a de menor custo
Melhor = SelecionaMenor(Custo,1);
// Imprime a melhor rota
printf("\nA melhor rota é");
PrintMatrix("Rota",[1 Rotas(Melhor,:) 1]);
printf("com custo total = %d.",Custo(Melhor));
```

Figura 418: O programa CaixeiroViajante.sce

A Figura 418 mostra o programa completo, que também usa as funções **SelecionaMenor** (Figura 345, página 213) e **PrintMatrix** (Figura 346, página 213). Executando este programa com o arquivo Distancias.txt (Figura 415), nós vemos que a melhor rota é [1 2 3 5 4 1] com custo total = 15, mostrada na Figura 409.

Muito bem, temos um programa que resolve o problema do caixeiro viajante. Mas será que com ele nós poderemos encontrar o melhor roteiro para visitar de avião todas as 27 capitais brasileiras? É melhor nem tentar. Com n cidades, temos $(n-1)!$ permutações a explorar. Para o exemplo com 5 cidades, são $4! = 24$ possibilidades, fácil. Para 8 cidades, temos $7! = 5040$ rotas a examinar, sem problemas, mas para as capitais brasileiras, são $26! \cong 4 \times 10^{26}$ permutações a serem examinadas!

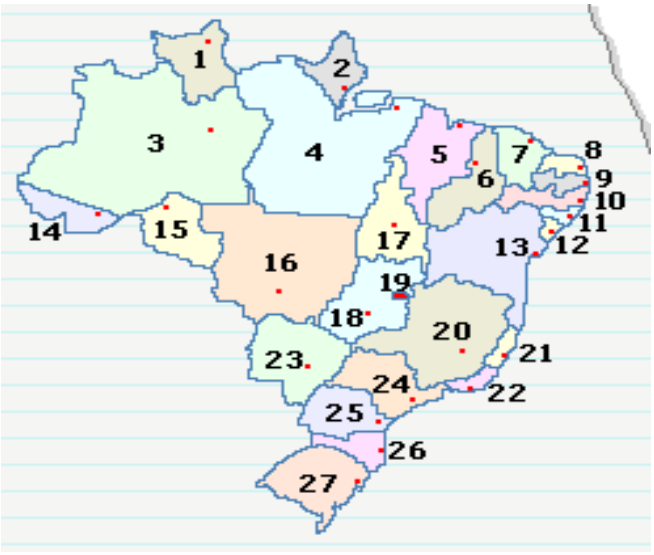


Figura 419: Qual é o melhor roteiro para visitar as 27 capitais brasileiras?

Mas isso não seria simplesmente um defeito do nosso algoritmo, cuja complexidade é $O(n!)$? Repetindo a pergunta feita para os algoritmos de ordenação, seria possível construir um algoritmo com complexidade polinomial para o caixeiro viajante? Efetivamente existem soluções melhores, com técnicas mais sofisticadas que reduzem significativamente o número de roteiros a serem examinados. Mas não reduzem tanto. A melhor solução já encontrada[53] tem complexidade $O(n^2 2^n)$, o que para n suficientemente grande cresce mais rapidamente do que qualquer potência de n, ou do que qualquer polinômio em n.

Senão, podemos provar que não existe tal solução? Infelizmente não temos uma boa resposta para estas duas questões. Após décadas de intensa pesquisa, não se conhecem algoritmos com complexidade polinomial para este problema, mas ninguém conseguiu uma prova de que não existem.

Cientistas da computação classificam problemas e algoritmos conforme sua ordem de complexidade:

- Problemas com complexidade polinomial – cujo tempo cresce com alguma potência de n – são enquadrados na classe P, e são considerados “educados”.
- Problemas cuja complexidade cresce com o tamanho da entrada mais rapidamente do que qualquer polinômio são chamados “intratáveis”.
- Dentre os intratáveis, a classe NP compreende os problemas onde, dada uma resposta, pode-se verificar se ela é ou não uma solução em tempo polinomial. Este é o caso do problema do caixeiro viajante, pois se alguém afirma ter encontrado uma rota com custo C, esta afirmativa pode ser verificada com facilidade, somando-se os custos das etapas na rota proposta. NP não significa Não Polinomial, mas Não determinístico Polinomial. A idéia é que os problemas em NP são resolvidos em tempo polinomial por uma algoritmo não determinístico, que em uma fase inicial “adivinha” uma solução, que é depois testada.
- Um subconjunto de NP é a classe dos problemas NP-completos, que são tais que qualquer problema em NP pode ser transformado em uma de suas instâncias.

Outro exemplo de problema NP-completo é o problema da mochila, que consiste em descobrir qual é a melhor escolha – a de maior valor total – de objetos com pesos e valores a serem colocados em uma mochila que tem uma capacidade máxima que não pode ser ultrapassada.

[53] “Travelling salesman problem - Wikipedia, the free encyclopedia”, acessado 11 de dezembro de 2012, http://en.wikipedia.org/wiki/Travelling_salesman_problem#Computational_complexity.

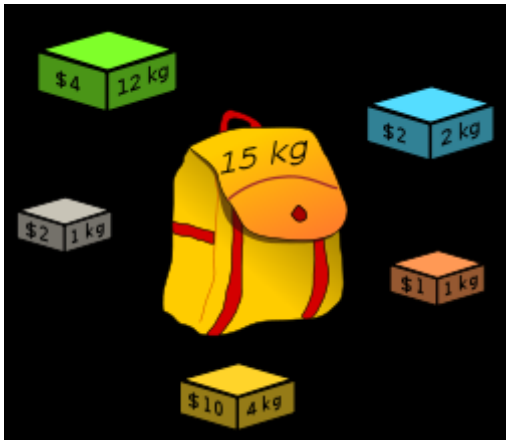


Figura 420: Quais objetos devem ser colocados na mochila para se obter o maior peso total sem ultrapassar a capacidade da mochila?

Se você conseguir desenvolver um algoritmo com complexidade polinomial para resolver o problema do caixeiro viajante ou o da mochila, ou provar que não existe tal solução, você ficará famoso, e ganhará um prêmio de um milhão de dólares!

12.3 Problemas indecidíveis: O Problema da Correspondência de Post

Vamos agora estudar um problema proposto pelo matemático Emil Post em 1946. Considere um estoque ilimitado de dominós de um número finito de tipos.

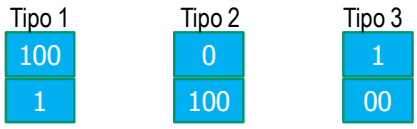


Figura 421: Exemplo de dominós para o problema da Correspondência de Post

Todos os dominós de um tipo têm um mesmo string de 0s e 1s na parte de cima e outro na parte de baixo. O problema da correspondência de Post é resolvido se você encontrar uma seqüência de dominós tal que os strings formados pela concatenação dos strings superiores e inferiores sejam iguais. A Figura 422 mostra uma solução para o problema proposto na Figura 421.

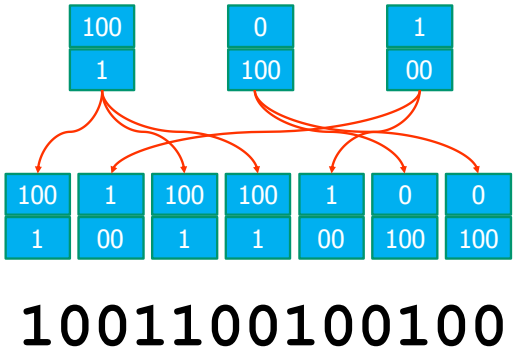


Figura 422: Uma solução com 7 dominós

Vamos representar uma sequência de dominós por um vetor de índices como [1 3 1 1 3 2 2], que significa “um dominó do tipo 1, seguido de um dominó do tipo 3, seguido de dois do tipo 1, seguido de um dominó do tipo 2, seguido por dois do tipo 2”.

Para resolver o problema de Post nós vamos novamente usar um algoritmo força-bruta, que gera todas as sequências possíveis de dominós, começando pelas menores. Se tivermos 3 tipos

de dominós, as primeiras sequências geradas são [1], [2], [3], [1 1], [1 2], [1 3], [2 1], [2 2], [2 3], [3 1], [3 2], [3 3], [1 1 1], [1 1 2], [1 1 3], [1 2 1], e assim por diante. Você pode ver que a geração das sequências pode ser feita por um processo equivalente à contagem em um sistema de base 3, sem o algarismo 0 e incluindo o algarismo 3.

```
exec("ReadPostProblem.sci");
exec("Sucessor.sci");
exec("strPost.sci");
exec("WritePostSolution.sci");
exec("WritePostProblem.sci");

// Leitura do problema de Post
[nTypes Up Down] = ReadPostProblem();
WritePostProblem(Up,Down);

found = %f;
seq = [];
while ~found
  seq = Sucessor(seq,nTypes);
  upString = strPost(Up,seq);
  dnString = strPost(Down,seq);
  found = upString == dnString;
  if found then
    WritePostSolution(seq,Up,Down);
  end
end
```

Figura 423: O programa Post.sce

A Figura 423 mostra o programa **Post.sce**, aonde:

- A função **ReadPostProblem** é usada para ler um problema de Post – um conjunto de tipos de dominós. Essa função retorna um inteiro **nTypes**, o número de tipos de dominós, e os vetores de strings **Up** e **Down**, que conterão respectivamente os strings da parte de cima e da parte de baixo de cada tipo de dominó no problema lido;
- A função **WritePostProblem** imprime na tela os dados do problema lido;
- O programa executa um loop que explora todas as seqüências de dominós extraídos dos tipos lidos, parando se encontrar uma seqüência onde as concatenações dos strings da parte superior e da parte inferior dos dominós são iguais.
- A variável **seq** contém uma seqüência de dominós, que a cada passagem do loop é substituída por sua sucessora, usando a função **Sucessor**;
- A função **WritePostSolution** é usada para imprimir na tela uma solução eventualmente encontrada;
- A função **strPost** constrói um string concatenando segundo uma sequência dada os strings na parte superior ou na parte inferior dos dominós.

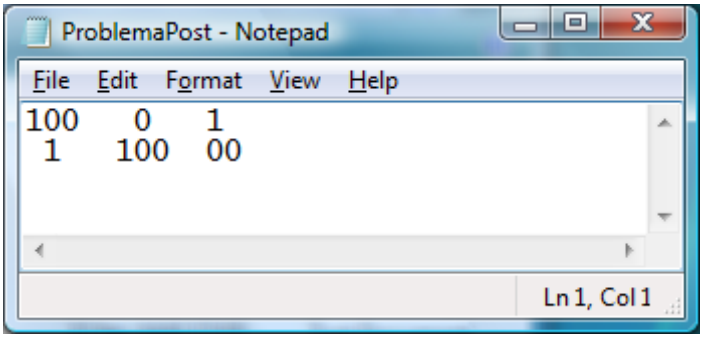


Figura 424: Tela do Bloco de Notas com um arquivo de tipos de dominós para o problema de Post da Figura 421

Vamos começar pela leitura de um conjunto de tipos de dominós. A Figura 424 ilustra o formato que escolhemos para arquivos com tipos de dominós que descrevem um problema de Post.

```
function [nTypes, Up, Down] = ReadPostProblem()
  PostFile = uigetfile("*.txt",pwd(),"Problema de Post");
  da = mopen(PostFile,"r");
  Lines = mgetl(da);
  Up = tokens(Lines(1));
  Down = tokens(Lines(2));
  [nTypes,nc]=size(Up);
Endfunction
```

Figura 425: A função ReadPostProblem

A função **ReadPostProblem** (Figura 425) faz uso da função **tokens**, fornecida pelo Scilab, que recebe um string como parâmetro de entrada, e produz um vetor coluna, onde os elementos do vetor são strings que, no string de entrada, estão separados por brancos ou tabulações.

```
function WritePostProblem(Up,Down);
  printf("\nProblema de Post:\n");
  [nl,nc] = size(Up);
  for i = 1:nl
    printf("%5s",Up(i));
  end
  printf("\n");
  for i = 1:nl
    printf("%5s",Down(i));
  end
endfunction
```

Figura 426: A função WritePostProblem

A função **WritePostProblem** (Figura 426) imprime na tela um problema armazenado nos vetores de strings **Up** e **Down**.

```
function s = strPost(Strs,seq)
  s = "";
  for i = 1:length(seq)
    s = s + Strs(seq(i));
  end
endfunction
```

Figura 427: A função strPost

A Figura 427 mostra a função **strPost** que constrói a concatenação dos strings em uma sequência de dominós.

```
function s = Sucessor(r,nTypes)
  // retorna o sucessor da sequência r
  VaiUm = 1;
  for i = length(r):-1:1
    if VaiUm > 0 then
      if r(i) < nTypes then
        r(i) = r(i) + 1;
        VaiUm = 0;
      else
        r(i) = 1;
      end
    end
  end
  if VaiUm == 1 then
    s = [1 r];
  else
    s = r;
  end
endfunction
```

Figura 428: A função Sucessor

A função **Sucessor** (Figura 428) gera, a partir de uma seqüência de dominós, a seqüência seguinte no processo de contagem. Essencialmente ela soma 1 ao número formado pelos "algarismos" que compõem a seqüência. A Figura 429 mostra alguns exemplos de uso desta função.

```
-->Sucessor([],3)
 ans  =
    1.

-->Sucessor([2 3 1],3)
 ans  =
    2.    3.    2.

-->Sucessor([3 3 3],3)
 ans  =
    1.    1.    1.    1.
```

Figura 429: Exemplos de uso da função Sucessor

Finalmente temos a função **WritePostSolution** (Figura 430) que imprime na tela a solução encontrada, em um formato tabular.

```
function WritePostSolution(seq, Up, Down)
  printf("\nSolução: %s\n",strPost(Up,seq));
  for i = 1:length(seq)
    printf("%5d",seq(i))
  end
  printf("\n");
  for i = 1:length(seq)
    printf("%5s",Up(seq(i)));
  end
  printf("\n");
  for i = 1:length(seq)
    printf("%5s",Down(seq(i)));
  end
endfunction
```

Figura 430: A função WritePostSolution

Já podemos testar o nosso programa para ver se encontramos a solução da Figura 422. Escolhendo como entrada o arquivo ProblemaPost.txt, vemos que o programa Post.sce efetivamente resolve este problema, produzindo a saída mostrada na Figura 431.

```
Problema de Post:
  100    0    1
    1  100   00
Solução: 1001100100100
    1    3    1    1    3    2    2
  100    1  100  100    1    0    0
    1   00    1    1   00  100  100
```

Figura 431: Saída do programa Post.sce, alimentado com o arquivo da Figura 424.

Como você já deve esperar, inferir desse primeiro sucesso que o nosso algoritmo resolve qualquer problema de Post é ingenuidade.

Figura 432: A menor solução para este problema de Post é uma sequência de 206 dominós!

Usando algoritmos mais sofisticados, é possível mostrar que a menor seqüência que resolve o problema da Figura 432 é formada por 206 dominós. Baseando-se em testes feitos pelo autor destas linhas, o tempo estimado para o programa Post.sce resolver este problema com um notebook seria de 10^{112} anos!

Poderíamos pensar que estamos diante de um problema como o do caixeiro viajante, mas mesmo isso é otimismo. A correspondência de Post pertence a uma classe de problemas chamados *indecidíveis*. Enquanto em problemas np-completos o espaço de busca cresce explosivamente com o tamanho da entrada, em problemas indecidíveis o espaço de busca é simplesmente ilimitado. Traduzindo para o problema de Post, o fato de não existir nenhuma seqüência de tamanho n que resolva um dado problema não quer dizer que não existam soluções de tamanho maior que n.

A indecidibilidade se refere ao caso geral, e não a instâncias particulares. Para algumas instâncias, como nos casos da Figura 422 e da Figura 432, pode ser possível encontrar uma solução. Para outras, pode ser possível demonstrar que não existe nenhuma solução, como seria o caso de um problema onde em todos os dominós o string da parte superior fosse mais longo que o da parte inferior.

Figura 433: Um problema de Post sem solução conhecida.

Resultados teóricos provam que não existe nenhum algoritmo que, para qualquer instância de um problema de Post, consiga decidir se existe ou não uma solução. A Figura 433 mostra um problema de Post para o qual não foi possível, até hoje, nem encontrar uma solução, e nem tampouco provar que ele não admite nenhuma solução.

13 Próximos Passos

Chegamos ao fim de nosso curso, onde vimos alguns dos conceitos fundamentais da ciência da computação. É hora de rever brevemente estas idéias para obter uma visão de conjunto, e também para apontar algumas direções para próximos passos.

- Computadores trabalham com informação, que pode ser digital (simbólica) ou analógica;
- Um processador trabalha essencialmente com informação simbólica, usando apenas dois símbolos, comumente notados 0 e 1; equipamentos de entrada e saída utilizam transformações análogo-digital ou digital-analógica quando conveniente.
- Um bit é a unidade de memória capaz de armazenar um destes símbolos. Com n bits, pode-se representar 2^n coisas distintas;
- Um código é uma convenção para a interpretação de conjuntos de bits. Códigos importantes incluem ASCII e UTF-8, para a representação de caracteres, binários sem sinal, binários em complemento de 2 para incluir também números negativos, e ponto flutuante.
- As operações booleanas NOT, AND e OR realizam transformações muito simples de bits, mas tudo o que um computador faz é através da composição destas operações;
- Transistores podem ser utilizados para implantar circuitos chamados portas lógicas, que realizam as operações booleanas. Transistores são implantados de forma extremamente compacta em semi-condutores, e realizam as operações booleanas muito rapidamente.
- Portas lógicas podem teoricamente realizar qualquer transformação de informação; colocando a informação de entrada e a de saída codificadas em uma tabela da verdade, podemos construir um circuito que realiza a transformação desejada.
- Isso funciona perfeitamente para circuitos pequenos, como para a construção de um circuito de soma completa, capaz de somar duas variáveis de um bit.
- O uso direto de portas lógicas para transformação de informação é entretanto limitado por razões práticas. Para a soma de dois inteiros de 32 bits, teríamos uma tabela da verdade com $2^{64} \cong 1{,}8 \times 10^{10}$. Um supercomputador que gastasse um nanosegundo (10^{-9} segundos) para processar cada entrada da tabela da verdade demoraria 585 anos para terminar o processamento.
- Por sua vez, circuitos de soma completa podem ser ligados em em cascata, em um arranjo que permite a soma de variáveis de, digamos, 32 bits cada uma.
- Isto funciona para inteiros de 64 ou de 128 bits, mas dificilmente alguém pensaria em construir um circuito para calcular a soma de 20 números de 32 bits cada um.
- Com registradores, barramentos, unidade lógico-aritmética e memória dispostos em um arranjo adequado, podemos usar sinais de controle para guiar o fluxo de dados e obter a soma de 20 ou mais números de 32 bits, usando um acumulador e realizando uma soma de cada vez.
- O próximo e enorme passo é a automação da emissão dos sinais de controle, com o uso de um programa armazenado na memória, composto por instruções que são interpretadas – executadas – por uma unidade central de processamento,
- Um computador é portanto um circuito que transforma informação, que entretanto difere do circuito de soma porque a transformação realizada não é fixa, mas ditada por outra informação – o programa armazenado.

- Trocando o programa, trocamos a transformação realizada. O ganho em fllexibilidade é enorme, quando comparado com a construção de um circuito. Software é macio; hardware é duro.
- E programas podem conter loops, o que nos permite por exemplo calcular a soma de 50.000 ou mais números de 32 bits – façanha absolutamente impraticável para um circuito combinatório, que para isso deveria ter $50.000 \times 32 = 3.200.000$ bits de entrada.
- A construção de um programa mesmo pequeno em linguagem de máquina é uma tarefa infernal, mas uma das principais utilidades dos computadores é facilitar a construção de programas para computadores.
- Montadores ou assemblers são programas que permitem o uso de mnemônicos para a designar instruções e posições de memória. Mesmo sendo um avanço sobre a programação direta em linguagem de máquina, a programação que se consegue com estes sistemas ainda é muito detalhada, presa a uma arquitetura específica, sendo propensa a erros e sem portabilidade.
- Compiladores e interpretadores são também programas, que têm como entrada programas escritos em uma linguagem de alto nível, como Fortran, C ou Scilab, e que ou bem transformam estes programas em instruções de máquina a serem executados diretamente por um computador, ou – como é o caso do Scilab – têm internamente uma máquina virtual, que interpreta o programa recebido como entrada.
- Linguagens de alto nível oferecem abstrações que nos permitem escrever programas – descrições de transformação de informação – de uma forma muito mais próxima do nosso raciocínio.
- O Scilab, em particular, nos permite guardar valores em variáveis com um nome que podemos escolher. Esses valores podem ser numéricos, caracteres, ou lógicos.
- Variáveis Scilab são sempre matrizes; a linguagem oferece notações para designação de partes de uma matriz.
- Variáveis, constantes, chamadas de funções, parênteses e operadores podem ser combinados em expressões que resultam em valores Scilab, e que podem ser empregados em comandos de atribuição para alterar valores de variáveis.
- A linguagem Scilab oferece também comandos de controle do fluxo de execução, como o comando condicional **`if-then-else`**, e os loops **`while`** e **`for`**.
- Temos comandos de entrada e saída, como **`input`** e **`printf`**, e comandos para o tratamento de arquivos, como **`mopen`** e **`mclose`**, **`mgetl`**, **`fscanfMat`** e **`fprintfMat`**, que nos permitem usar armazenamento estável para massas de dados potencialmente grandes.
- Um programa Scilab é formado por um programa principal e por definições de funções Scilab. Programas e funções Scilab são armazenados em arquivos.
- Uma função Scilab define parâmetros formais de entrada e de saída. A chamada de uma função Scilab define parâmetros reais de entrada, que são expressões Scilab, e parâmetros reais de saída, que são variáveis que recebem os valores calculados pela função para seus parâmetros formais de saída.
- Funções Scilab podem conter comandos Scilab, variáveis locais e chamadas de funções – incluindo possivelmente chamadas à própria função, em um arranjo recursivo.
- A recursividade pode simplificar muito o desenvolvimento de algoritmos, pois expressa de forma natural o seu comportamento.
- Funções são uma importante ferramenta de modularização. Seu uso permite o desenvolvimento seja em momentos separados, seja por pessoas diferentes.
- Com o domínio de uma linguagem de programação nós pudemos atacar problemas de transformação de informação muito mais elaborados.

- Problemas de transformação de informação são em princípio resolvidos por algoritmos – métodos que prescrevem sequências de transformações elementares, e que são convenientemente implantados por programas de computadores – no nosso caso, programas Scilab.
- Usando a linguagem Scilab nós vimos diversos algoritmos para solução de problemas como leitura, processamento e escrita de dados, usando inicialmente teclado e monitor, para pequenos volumes de dados, e depois arquivos, para grandes volumes de dados.
- Para dois problemas clássicos de transformação de informação nós vimos diversas soluções: a pesquisa por um valor em um vetor, e a ordenação de um vetor.
- Nós vimos que algoritmos podem diferir – e muito – em sua eficiência no uso de recursos computacionais, como tempo de execução ou quantidade de memória.
- O termo complexidade computacional de um algoritmo é empregado para caracterizar suas exigências destes recursos como uma função do tamanho dos dados de entrada.
- Alguns problemas de transformação de informação têm limites inferiores para a complexidade de qualquer algoritmo que o resolva. A ordenação de um vetor, por exemplo, é no melhor caso $O(n \log(n))$.
- Acredita-se que o limite inferior para uma classe de problemas conhecida como np-completos, como o problema do caixeiro viajante, tem complexidade intrínseca crescente em taxa maior que qualquer polinômio em n.
- Outros problemas são ainda piores. Para problemas chamados indecidíveis, como o da correspondência de Post, não existem algoritmos com um tempo limite garantido para qualquer entrada.

Há muito o que se aprender em computação, pura ou aplicada às ciências e às engenharias, e diversas outras disciplinas podem extender o seu conhecimento nesta área:

- *Organização de Computadores.* O projeto de sistemas digitais e de computadores é uma vasta área, que é normalmente estudada em disciplinas como Sistemas Lógicos, Organização de Computadores, ou Arquitetura de Computadores. Metodologias para o projeto e implantação de circuitos digitais são vistas com maior profundidade, assim como aspectos teóricos.
- *Cálculo Numérico*. Nessa disciplina são vistos, como o nome indica, algoritmos para a solução de problemas numéricos, de grande importância para todos os cientistas e engenheiros. Tipicamente são vistos algoritmos para encontrar zeros (raízes) de funções (dos quais o método da bisseção visto na Seção 11.2 é um exemplo), para a solução de sistemas de equações lineares, para interpolação, para solução de equações diferenciais, para integração numérica, e vários outros. A preocupação com erros de arredondamento e de truncamento, e com a sua propagação é tratada com muito maior profundidade.
- *Algoritmos e Estruturas de Dados*. Esta área trata extensamente de algoritmos para ordenação e pesquisa (dos quais nós vimos alguns exemplos), do uso de estruturas de dados mais flexíveis que matrizes, como listas, árvores e grafos, do casamento de padrões, sempre com um tratamento bem mais rigoroso dos aspectos relacionados à complexidade.
- *Programação Orientada a Objetos*. A programação orientada a objetos, ou POO, oferece estruturas linguísticas para uma definição elegante de dados e de formas de interação. A programação torna-se mais compacta e mais segura, permitindo um intenso reaproveitamento de código. Nenhum programador mais sério pode se permitir desconhecer a POO, que foi introduzida já em 1967 com a linguagem Simula, e depois explorada com Smalltalk. Estas duas linguagens ainda sobrevivem, mas o uso de C++ e de Java é hoje muito maior.

- *Bancos de Dados*. Bancos de dados são sistemas de armazenamento que extendem muito o conceito de arquivos. Um SGBD (Sistema de Gerência de Banco de Dados) como Oracle, PostGresSQL, MySQL, SQL Server, e outros, permite a recuperação de dados por perguntas (*queries*, em inglês. SQL quer dizer *Standard Query Language*) como " me dê a lista dos alunos de engenharia civil com idade entre 18 e 20 anos que já tenham cursado Programação de Computadores ou Cálculo Numérico". SGBDs tratam também do controle de concorrência, controlando o acesso simultâneo a uma mesma base por diversos usuários, e garantindo a preservação de sua integridade.
- *Sistemas Reativos*. São sistemas que reagem a diversos estímulos de forma a, por exemplo, controlar um alto-forno, aumentando a combustão ao perceber uma baixa de temperatura, e diminuindo quando a temperatura do forno está alta. Sistemas operacionais como o Windows ou o Linux são também exemplos de sistemas reativos, controlando os estímulos recebidos pelos equipamentos de entrada e saída. Engenheiros e cientistas trabalham normalmente com sistemas reativos menores, controlando máquinas e equipamentos de laboratórios.

Para finalizar, alguns conselhos.

- O Scilab é adequado para o desenvolvimento de pequenos programas voltados para ciências e engenharias. Para programas maiores, com mais de 1000 linhas, considere o uso de outras linguagens, com C, Fortran, C++ ou Java.
- Qualquer que seja a linguagem escolhida, procure usar bibliotecas de funções desenvolvidas por profissionais. Só desenvolva o que for realmente necessário. Funções de boas bibliotecas têm código mais robusto e mais rápido, e dão tratamento adequado a erros numéricos.
- Experimente sempre! Enquanto você não estiver fazendo programas para uso em produção, errar não machuca. Dificilmente um sistema se estraga por um erro de programação, e a experimentação é essencial para o aprendizado.

***

Nós iniciamos este texto enumerando áreas do conhecimento que foram ajudadas pela computação, mas vamos terminar com uma confissão. A área que mais se beneficia com os resultados da Ciência da Computação é ... a própria computação. Computadores auxiliam a programar computadores, a construir circuitos que são usados por computadores, a projetar novas linguagens de programação, e também a construir programas que auxiliam o aprendizado de computação, como você experimentou ao longo deste curso. É este poderoso processo de auto-ajuda faz da computação o impulsor das mudanças de imenso impacto em nossas vidas.

Índice Remissivo

%eps, 124
%pi, 124
.sce, 132
ABus, 282
acumulador, 91
Álgebra Booleana, 47
algoritmo, 198
ALU, 93
ambiente Scilab, 121
AND, 48
Aritmética matricial, 148
arquivo-programa, 132
árvore binária, 244
ASCII, 42
Assembler, 272
assembly, 272
atuadores, 29
barramento, 88
Basic, 119
Binários sem Sinal, 44
bit, 22
bloco "então", 141
bloco "senão", 141
Blue Gene, 13
bootstrap, 104
bps, 30
C, 119
C++, 119
Carta de tempo, 85
chamadas da função, 186
ciclo de instrução, 272
Ciclo de Micro-Instrução, 287
Circuito para sincronização, 281
circuito principal da CPU Pipoca, 279
Cleve Moler, 121
clock, 94
Cobertura dos 1s, 64
Cobol, 119
Codificação com Deslocamento, 45
código da instrução, 275
Comandos Aninhados, 173
comentários, 133
comparação de binários sem sinal, 78
comparador de 1 bit, 78
compilador, 118
Complemento de 2, 45
complexidade computacional, 199
complexidade linear, 207
condutor perfeito, 53
console do Scilab, 122
Construindo matrizes, 152
controlled buffer, 88
conversão binário-decimal, 44
conversões A/D, 23
conversões D/A, 23
Correção, 198
data width, 87
DBus, 282
Debug, 282
Demultiplexadores, 81
desenvolvimento *top-down*, 193
diretório corrente, 133
dividir para conquistar, 225
Dvorak, 29
eco, 123
  supressão do eco, 124
Eficiência, 198
else, 141
endereço, 89
endfunction, 187
escopo de variáveis, 188
Especificação, 198
estouro, 67
Expressões booleanas, 48
expressões lógicas, 138
eye, 153
fatoração de números inteiros, 200
fetch, 292
flip-flop tipo D, 84
Flip-flops, 84
folhas, 244
Fortran, 118
fprintfMat, 179
fscanfMat, 179
função recursiva, 195
Funções, 185
function, 187
George Boole, 47
getf, 189
GetOperand, 293
IBM PC, 12
IEEE 754, 46
if, 141
Indentação, 175

Informação, 18
Informação analógica, 18
informação digital, 18
input, 136
INPUT, 282
instruções, 274
instruções de desvio, 274
instruções de máquina, 272
Instruction Register, 273
Integração por Trapézios, 228
Internet, 17
interpretador, 121
IR, 273
isolante perfeito, 53
Java, 119
jsr, 285
jump to subroutine, 285
kiss principle, 209
lâmpada de 7 segmentos, 75
largura de bits, 87
leg, 181
legibilidade, 174
limites fisiológicos, 28
linspace, 152
LISP, 120
loaders, 277
mantissa, 46
MAR, 90
Matlab, 121
matriz identidade, 153
matriz inversa, 150
matriz transposta, 150
Matrizes, 145
Matrizes de Strings, 159
meia-soma, 64
Memórias, 31
Memórias secundárias, 33
Memórias terciárias, 33
Memory Addres Register. *See* MAR
merge, 215
mgetl, 178
micro Instruction Register, 284
micro Program Counter, 284
micro-assembler, 292
microinstruções de desvio, 285
microinstruções de sinal, 284
Micro-Programa, 292
microprogramação, 284
mIR, 284
mnemônicos, 276
modo de endereçamento, 275
montador, 279
montagem, 272
mPC, 284
Multiplexadores, 81
NAN, 47
NAND, 51
nomes de variáveis, 123
NOR, 51
NOT, 48
ones, 153
operadores relacionais, 138
operando, 275
OR, 48
ordenação, 210
ordenação por Intercalação, 215
Ordenação por Seleção e Troca, 210
oscilador, 94
Ou Exclusivo. *See* XOR
OUTPUT, 282
overflow, 67
palavras, 89
Parâmetros formais, 187
parâmetros reais, 188
Pascal, 119
PC, 273
Pesquisa Binária, 207
Pesquisa Seqüencial, 207
PHP, 119
pilha, 196
Planilha Pipoca.xls, 295
plot2d, 154
polígono, 181
Ponto Flutuante, 46
printf, 141
problema de transformação de informação, 198
processador, 21
produto elemento a elemento, 149
produto matricial, 149
profundidade, 244
Program Counter, 273
programa, 271
Programa em Execução, 277
programa executável, 276
Programa Executável, 277
Programa Fonte, 277
programa principal, 186
Prolog, 120
prova formal, 198
pwd, 178
Python, 119
raiz da árvore, 244
RAM, 31

rand, 153
rect, 181
registrador, 86
registrador circular, 95
Registradores, 31
retorno da função, 186
return, 285
Return Address, 284
RGB, 44
RoadRunner, 13
ROM, 89
sci, 189
Scilab, 121
SciPad, 132
seleção e troca, 211
Select Sort, 210
sensores, 29
SetInEmpty, 281
SetInFull, 281
Signals, 284
Sinal e Amplitude, 45
Síntese de Circuitos Combinatórios, 69
size, 146
software, 271
soma canônica, 64
soma de Riemann, 228
soma-completa, 67
somador de n bits, 67
splitters, 87
Strings, 139
sub-circuito, 72
SumX, 277
supercomputador, 14
tabela da verdade, 62
teclado Dvorak, 29
testes, 199
then, 141
timer, 200
Transistores, 50
Trocar os valores de duas variáveis, 212
uigetfile, 177
unidade de controle, 274
unidade lógico-aritmética, 93
vai-um, 66
Variáveis, 123
variáveis locais, 188
Variáveis Lógicas, 143
variável, 122
vem-um, 67
Vetores, 146
volátil, 31
volatilidade, 31
Von Neumann, 272
XOR, 56
xtitle, 181
zeros, 153

Referências

Andrews, D. (n.d.). *Primes R US*. Retrieved from http://www.geocities.com/primes_r_us/

Burch, C. (2002). *Logisim: A Graphical Tool for Designing and Simulating Logic Circuits*. Retrieved March 2009, from http://ozark.hendrix.edu/~burch/logisim/

Dijkstra, E. W. (1972). Chapter I Notes on Structured Programming. In E. W. O. J. Dahl, *Structured Programming.* Eds. ACM Classic Books Series. Academic Press Ltd., London, UK, 1-82.

Flickr. (n.d.). Retrieved Fevereiro 2010, from http://farm4.static.flickr.com/3444/3348244651_fef16ef641.jpg

Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York, USA: W. H. Freeman & Co.

Hollasch, S. (2005). *IEEE Standard 754 Floating Point Numbers*. Retrieved August 25, 2009, from http://steve.hollasch.net/cgindex/coding/ieeefloat.html

HotHardware.com. (n.d.). *Intel Core 2 Extreme QX9770 Performance Preview*. Retrieved Fevereiro 2010, from http://hothardware.com/articles/Intel_Core_2_Extreme_QX9770_Performance_Preview/

I.Ziller, J. B. (1954). *Preliminary Report: Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN.* International Business Machines, Applied Science Division.

IBM. (n.d.). *IBM Archives - Personal Computer*. Retrieved 2009, from http://www-03.ibm.com/ibm/history/exhibits/pc/pc_1.html

Lohninger, H. (2006). (Vienna University of Technology) Retrieved from http://www.vias.org/simulations/simusoft_adconversion.html

Lyon, B. (2005). *The Opte Project*. Retrieved August 2009, from http://opte.org/

Mathworks. (n.d.). Retrieved Fevereiro 2010, from http://www.mathworks.com/

McJones, P. (n.d.). *History of FORTRAN and FORTRAN II*. (Computer History Museum) Retrieved April 2009, from Software Preservation Group: http://www.softwarepreservation.org/projects/FORTRAN/

McKeeman, B. (n.d.). *MATLAB 101 - A talk for the MIT Computer Science Department*. Retrieved Fevereiro 2010, from http://www.cs.dartmouth.edu/~mckeeman/references/matlab101/matlab101.html

Morris, R. J. (2003). The Evolution of Storage Systems. *IBM Systems Journal, 42*(2), 205-217.

Neumann, J. v. (1945). *Michael D. Godfrey home page.* Retrieved Março 2010, from http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf

O'Reilly Media. (n.d.). *Language Poster*. Retrieved April 2009, from History of Programming Languages: http://oreilly.com/news/languageposter_0504.html

Scilab Consortium. (n.d.). *Scilab Home Page*. Retrieved from http://www.scilab.org/

StatLib. (1989). *StatLib - Datasets Archive*. (Department of Statistics, Carnegie Mellon University) Retrieved March 2009, from http://lib.stat.cmu.edu/datasets/

Top500. (n.d.). *Top 500 supercomputers*. Retrieved from http://www.top500.org

Wolffdata. (n.d.). *Wolffdata*. Retrieved 2008, from ScilabStarter: http://www.wolffdata.se/scilab/ScilabStarter.pdf

Referências Zotero

Alan M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. *Proceedings of the London Mathematical Society* s2–42, nº 1 (1937): 230–265.

“Alan Turing - Wikipedia, the free encyclopedia”. Acessado 11 de março de 2011. http://en.wikipedia.org/wiki/Alan_Turing.

“An investigation of the laws of thought [microform] : on which are founded the mathematical theories of logic and probabilities : Boole, George, 1815-1864 : Free Download & Streaming : Internet Archive”. Acessado 19 de agosto de 2011. http://www.archive.org/details/investigationofl00boolrich.

“Applications with Scilab, Maxima, Geogebra”. Acessado 28 de setembro de 2011. http://www.wolffdata.se/.

“ASCII - Wikipedia, the free encyclopedia”. Acessado 18 de agosto de 2011. http://en.wikipedia.org/wiki/ASCII.

“Barbara Blackburn, the World’s Fastest Typist”. Acessado 15 de setembro de 2012. http://rcranger.mysite.syr.edu/famhist/blackburn.htm.

Barret Lyon. “The Opte Project”. Acessado 7 de março de 2011. http://opte.org/.

bigbluesky2002. “YouTube - Swiss Antique Music Box”. Acessado 1 de abril de 2011. http://www.youtube.com/watch?v=-tzWt1X3oOg&p=66A384A130B0DCE4.

Bill McKeeman. “MATLAB 101 -- A talk for the MIT Computer Science Department”, novembro de 2005. http://www.cs.dartmouth.edu/~mckeeman/references/matlab101/matlab101.html.

Bryant, R.E., K. Sutner, e M.J. Stehlik. *Introductory Computer Science Education at Carnegie Mellon University: A Deans’ Perspective*. Tech. Report CMU-CS-10-140, Carnegie Mellon University, Pittsburgh, 2010, http://link. cs. cmu. edu/article. php, 2010.

Burch, Carl. “Logisim”. Acessado 20 de agosto de 2011. http://ozark.hendrix.edu/~burch/logisim/.

———. “Logisim - a graphical tool for designing and simulating logic circuits”. Acessado 7 de março de 2011. http://ozark.hendrix.edu/~burch/logisim/.

Chizmar, Jack. “The Effective Teaching and Learning Network - training course and information for teachers - TQM and Classroom Research”. Acessado 7 de março de 2011. http://www.etln.org.uk/resources/page16.html.

“Claude Shannon - Wikipedia, the free encyclopedia”. Acessado 19 de agosto de 2011. http://en.wikipedia.org/wiki/Claude_Shannon.

“Computer History Museum - Timeline of Computer History”. Acessado 16 de fevereiro de 2012. http://www.computerhistory.org/timeline/?year=1946.

"Download PSPad - free unicode developer editor, handles near any syntax like HTML, PHP, XHTML, JavaScript, ASP, Perl, C and many other languages with HEX editor, multilanguage interface". Acessado 6 de março de 2013. http://www.pspad.com/en/download.php.

"ENIAC - Wikipedia, the free encyclopedia". Acessado 16 de fevereiro de 2012. http://en.wikipedia.org/wiki/ENIAC.

"Euclidean algorithm - Wikipedia, the free encyclopedia". Acessado 8 de março de 2011. http://en.wikipedia.org/wiki/Euclidean_algorithm#cite_note-2.

"File:MagneticMedia.png - Wikipedia, the free encyclopedia". Acessado 12 de agosto de 2011. http://en.wikipedia.org/wiki/File:MagneticMedia.png.

"File:Pdp-11-70-panel.jpg - Wikipedia, the free encyclopedia". Acessado 2 de setembro de 2011. http://en.wikipedia.org/wiki/File:Pdp-11-70-panel.jpg.

"George Boole - Wikipedia, the free encyclopedia". Acessado 23 de fevereiro de 2012. http://en.wikipedia.org/wiki/George_Boole.

"Home | TOP500 Supercomputing Sites". Acessado 7 de março de 2011. http://www.top500.org/.

"IBM Archives: IBM Personal Computer". Acessado 7 de março de 2011. http://www-03.ibm.com/ibm/history/exhibits/pc/pc_1.html.

"IBM Card Sorters". Acessado 22 de fevereiro de 2012. http://www.columbia.edu/cu/computinghistory/sorter.html#references.

John W. Backus et al. "Preliminary Report: Specifications for the IBM Mathematical FORmula TRANSlating System, FORTRAN | Computer History Museum", 1954. http://www.computerhistory.org/collections/accession/102679231.

"Ken&Den picture". Acessado 24 de setembro de 2012. http://cm.bell-labs.com/cm/cs/who/dmr/picture.html.

Lawrence Livermore National Laboratory. "BlueGene/L Photo Gallery". Acessado 11 de março de 2011. https://asc.llnl.gov/computing_resources/bluegenel/photogallery.html.

M. R. Garey, e D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. First Edition. W. H. Freeman, [s.d.].

"Magnetic-core memory - Wikipedia, the free encyclopedia". Acessado 12 de março de 2011. http://en.wikipedia.org/wiki/Magnetic_core_memory.

Mathworks. "MathWorks - MATLAB and Simulink for Technical Computing". Acessado 7 de março de 2011. http://www.mathworks.com/.

"Moore's law - Wikipedia, the free encyclopedia". Acessado 3 de março de 2013. http://en.wikipedia.org/wiki/Moore_Law.

"Muḥammad ibn Mūsā al-Khwārizmī - Wikipedia, the free encyclopedia". Acessado 21 de agosto de 2012. http://en.wikipedia.org/wiki/Mu%E1%B8%A5ammad_ibn_M%C5%ABs%C4%81_al-Khw%C4%81rizm%C4%AB.

"NCAR's MSS exceeds 2 petabytes". Acessado 12 de agosto de 2011. http://www.cisl.ucar.edu/news/04/fotoweek/0729.mss2pb.html.

"Photographic film image". Acessado 7 de março de 2011. http://farm4.static.flickr.com/3444/3348244651_fef16ef641.jpg.

"Punched_card.jpg (imagem JPEG, 2406×1160 pixels) - Redimensionada (29%)". Acessado 22 de junho de 2013. https://upload.wikimedia.org/wikipedia/commons/f/f3/Punched_card.jpg.

"SciDAC Review - SIMULATION SCALE: In HPC Simulations, How Much is ENOUGH?" Acessado 2 de agosto de 2011. http://www.scidacreview.org/0801/html/scale.html.

Scilab Consortium. "Home - Scilab WebSite". Acessado 7 de março de 2011. http://www.scilab.org/.

"sciport-3.0.pdf". Acessado 7 de março de 2011. http://www.dca.ufrn.br/~pmotta/sciport-3.0.pdf.

Sosa, Carlos, e Knudson, Brant. *IBM System Blue Gene Solution: Blue Gene/P Application Development*. IBM Redbooks SG24-7287-03. Acessado 14 de fevereiro de 2012. http://www.redbooks.ibm.com/redbooks/pdfs/sg247287.pdf.

"Sound Transducer". Acessado 2 de agosto de 2011. http://www.electronics-tutorials.ws/io/io_8.html.

Steve Hollasch. "IEEE Standard 754 Floating-Point". Acessado 7 de março de 2011. http://steve.hollasch.net/cgindex/coding/ieeefloat.html.

"The History of Programming Languages - O'Reilly Media". Acessado 7 de março de 2011. http://oreilly.com/news/languageposter_0504.html.

"The Prime Pages (prime number research, records and resources)". Acessado 7 de março de 2011. http://primes.utm.edu/index.html.

"Travelling salesman problem - Wikipedia, the free encyclopedia". Acessado 11 de dezembro de 2012. http://en.wikipedia.org/wiki/Travelling_salesman_problem#Computational_complexity.

"Unicode - Wikipedia, the free encyclopedia". Acessado 19 de agosto de 2011. http://en.wikipedia.org/wiki/Unicode.

Apêndice A: A CPU Pipoca

Neste Apëndice nós apresentamos a CPU Pipoca, em um nível de profundidade destinado a professores e a alunos avançados. O circuito da Figura 135 tem os mesmos elementos básicos de uma CPU (veja também a Figura 434), com memória, barramentos e registradores controlados por sinais, e uma unidade lógico aritmética que realiza transformações como somas, subtrações, etc. Um circuito assim tem teoricamente a mesma capacidade de processamento (de transformação de informação) de um computador. Dizemos teoricamente porque é preciso que alguém se disponha a manipular os valores de entrada e os sinais de controle, como já fizemos nos exemplos de fluxos de dados já apresentados. Ou seja, usar esse circuito para fazer a soma de 500.000 números é possível, mas não é executável por um ser humano.

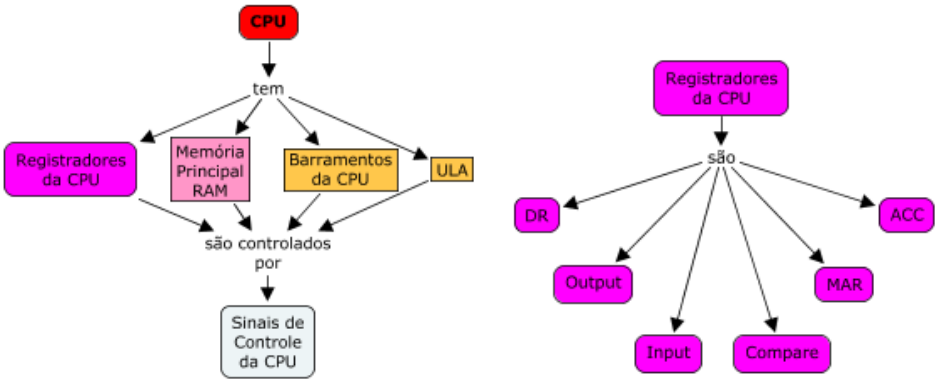


Figura 434: Elementos básicos para processamento e registradores da CPU Pipoca

A idéia central de um computador é a *automação da emissão de seqüências de sinais de controle* segundo o desejo de uma pessoa, desejo este expresso por um *programa*. Queremos que este circuito transforme informação de maneira flexível, com seu comportamento moldado por um programador. Trocando-se o programa, troca-se a transformação de informação produzida pelo circuito.

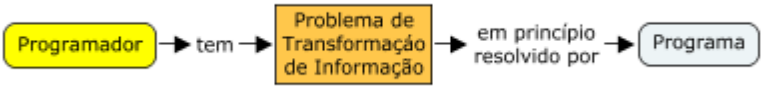


Figura 435: Um programa resolve um problema de transformação de informação

Considerando que queremos poder trocar com facilidade o programa que rege um computador, a única opção é colocar programas em uma memória, ao invés de implementá-los com circuitos. O termo software, com o prefixo *soft*, macio, foi um neologismo criado para se contrapor a hardware, mais duro, constituído por circuitos concretos. Software é muito mais flexível.

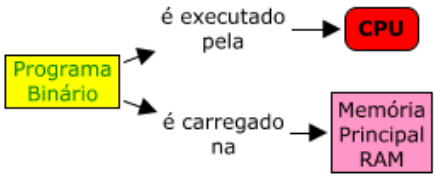


Figura 436: Um programa deve estar codificado em bits para ser carregado em alguma memória

Alguns dos primeiros computadores tinham uma memória para programas e outra para dados, mas na quase totalidade dos computadores atuais programas e dados são armazenados na mesma memória, seguindo a arquitetura chamada de Von Neumann, que propôs este arranjo em 1945 (Neumann, 1945).

Um programa tem um caráter dual, pois ele deve ao mesmo tempo:

- expressar a forma com que uma pessoa – um programador – pensa em resolver um problema de transformação de informação, e
- poder ser executado pelo processador, isto é, por um circuito digital.

Figura 437: Programa Fonte e Programa Binário

Na prática estas duas demandas podem ser atendidas da seguinte maneira. Um programa é composto por *instruções de máquina*, que são escritas por um programador em uma linguagem chamada *Assembler*, e depois "traduzidas" para serem executadas por uma CPU. Isso é feito por um processo que chamamos de montagem (*assembly*) das instruções.

Figura 438: Montagem de um programa

A.1 Ciclo de Instrução

Tendo um programa carregado na memória, um computador executa repetidamente, incansavelmente, um *ciclo de instrução*.

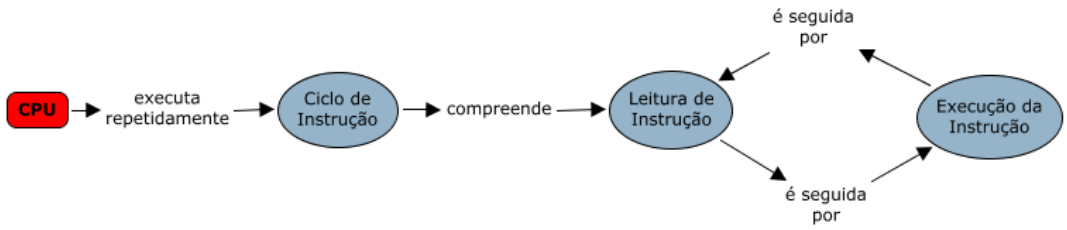


Figura 439: Ciclo de execução de instruções

Em cada ciclo uma instrução é lida da memória e executada. A execução da instrução:

- provoca alterações nos conteúdos da memória e/ou dos registradores, e
- determina o endereço da próxima instrução a ser executada.

A coordenação do ciclo de instrução exige novos registradores na CPU. Na Pipoca, estes são:

- *PC*, de *Program Counter*, que contém o endereço da próxima instrução a ser executada
- *IR*, de *Instruction Register*, que contém a instrução em execução
- *Sinc Entrada* e *Sinc Saída*, necessários para sincronizar operações de entrada e saída, que veremos mais tarde.

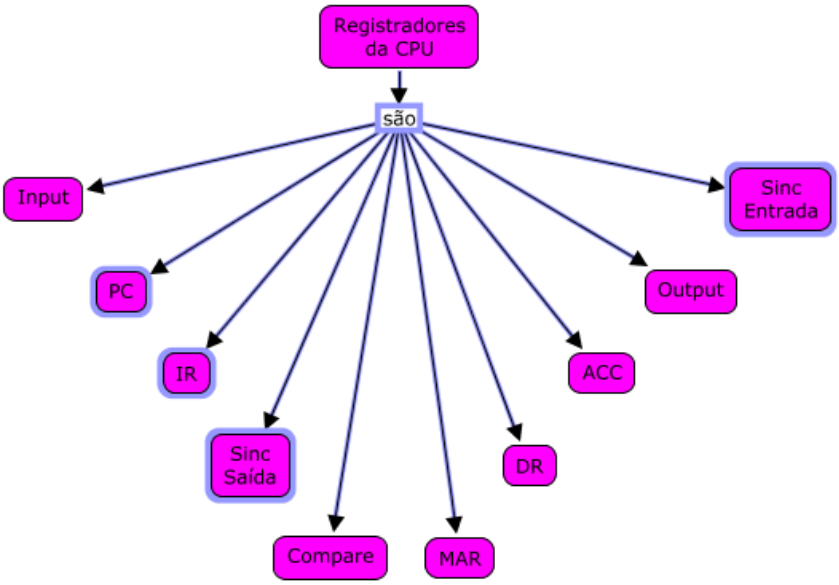


Figura 440: A coordenação da execução de instruções na Pipoca exige registradores extras: PC (Program Counter), IR (Instruction Register), Sinc Saída e Sinc Entrada

Um ciclo de instrução se inicia pela leitura da instrução (ou *instruction fetch*) a ser executada. A instrução é lida da memória, no endereço dado pelo conteúdo do registrador *PC*, e colocada no registrador *IR*.

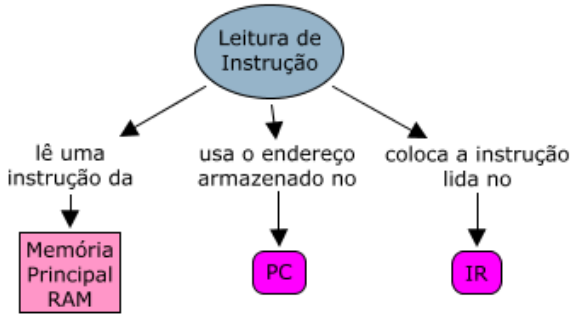


Figura 441: Leitura de uma instrução

A execução da instrução pode modificar o conteúdo da memória ou dos registradores, e determina um novo valor para o registrador *PC* – o que significa determinar qual será a próxima instrução a ser executada.

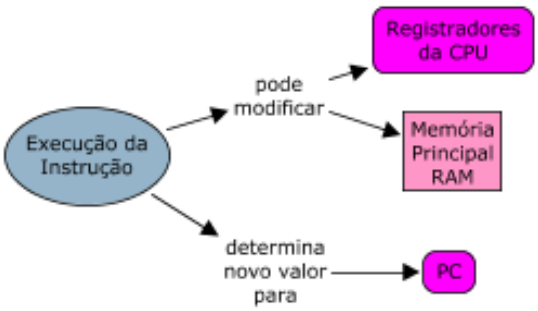


Figura 442: Execução de uma instrução

Processadores são construídos de forma tal que a próxima instrução a ser executada está normalmente no endereço na memória subseqüente ao ocupado pela instrução em execução. Ou seja, as instruções em princípio são executadas na mesma sequência em que se encontram armazenadas na memória. Instruções especiais, chamadas *instruções de desvio*, podem alterar essa ordem de execução. Desvios podem condicionais, dependendo por exemplo do resultado de uma comparação.

Um circuito chamado *unidade de controle* recebe como entradas o código da instrução, o status de registradores de comparações, um sinal de um oscilador (um clock), e se incumbe da geração correta dos sinais de controle correspondentes que comandam o ciclo de instrução na CPU.

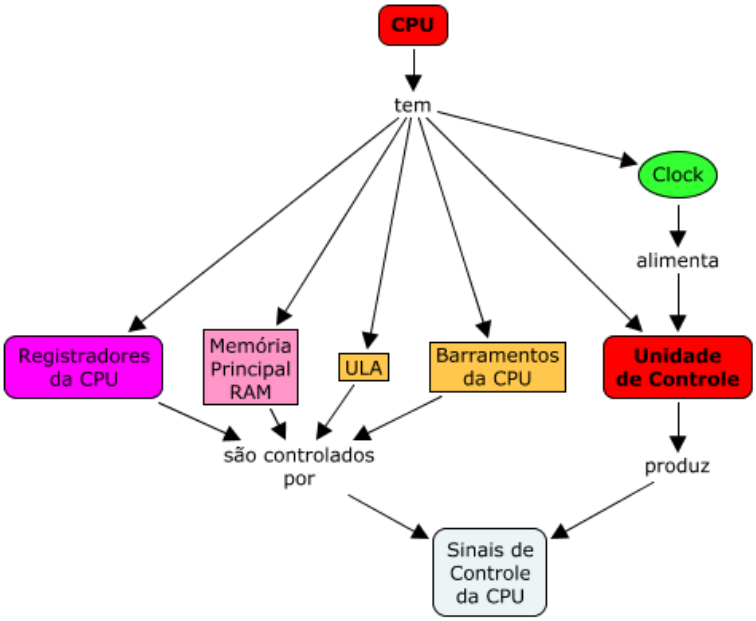


Figura 443: Uma unidade de controle e um clock são necessários para a execução de programas

1 Instruções

Para facilitar a programação, as instruções que constituem um programa não são estruturadas diretamente em termos de sinais de controle – o que entretanto se poderia esperar, visto que a emissão destes sinais é o efeito final necessário para se produzir uma computação. Ações típicas de instruções de máquina são “somar o conteúdo da posição X de memória ao acumulador”, ou “ler um dado em pinos de entrada e colocar na posição de memória Y”, com mais significado para o programador. A emissão efetiva dos sinais de controle fica definida

pelo que chamamos de *micro-instruções*. A execução de uma instrução é feita através da execução de várias micro-instruções, como veremos a seguir.

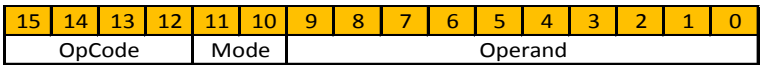


Figura 444: Formato de uma instrução da Pipoca

Nós vimos que instruções devem ser codificadas também em bits. A Figura 444 mostra o formato que escolhemos para as instruções da Pipoca. Cada instrução tem 16 bits, sendo 4 para o *código da instrução*, 2 para o *modo de endereçamento* (que explicaremos a seguir) e 10 para um *operando*.

Instruções são definidas em termos de alterações na visão que um programador tem da CPU. O conjunto completo de instruções da Pipoca está mostrado na Tabela 5. São 11 instruções ao todo. Como dispomos de 4 bits para o código da instrução, temos ainda espaço para quatro outras instruções, que podem vir a ser implementadas em novas versões.

Tabela 5: Conjunto de instruções da Pipoca

Description	Mnemonic	OpCode10	OpCode2	BranchAddr16
Adiciona o operando a *ACC,* deixando o resultado em *ACC*	ADD	0	0000	18
Compara o operando com *ACC* e coloca o resultado em *Compare*	COMPARE	1	0001	1C
Para a execução do programa	HALT	2	0010	45
Espera *InFull* = 1, e transfere o valor de *Input* para a palavra apontada pelo operando; faz *InFull* = 0	INPUT	3	0011	20
Desvia para a palavra apontada pelo operando	JMP	4	0100	27
Desvia para a palavra apontada pelo operando se "D=ACC" = 1	JMPEQ	5	0101	2B
Desvia para a palavra apontada pelo operando se "D>ACC" = 1	JMPGT	6	0110	2D
Desvia para a palavra apontada pelo operando se "D<ACC" = 1	JMPLT	7	0111	2F
Carrega o operando no acumulador	LOAD	8	1000	31
Espera *OutEmpty* = 1, e transfere o operando para o registrador *Output;* faz *OutEmpty* = 0	OUTPUT	9	1001	36
Transfere o valor de *ACC* para a palavra apontada pelo operando	STORE	10	1010	3B
Subtrai o operando de *ACC,* deixando o resultado em *ACC*	SUB	11	1011	41
Estes códigos podem ser usados para novas instruções		12	1100	
		13	1101	
		14	1110	
		15	1111	

Nessa tabela,

- o campo *Description* descreve o efeito da instrução;
- o campo *Mnemonic* contém códigos para cada instrução, que são empregados por um programador ao construir um programa;
- o campo *OpCode10* enumera as instruções e, com isso, fornece um código para cada uma delas;
- o campo *OpCode2* contém os mesmos valores de OpCode10, mas codificados em binário de 4 bits, destinados ao uso por computadores;
- O campo *BranchTable16* será explicado mais tarde.

Voltando à Figura 444, os dois bits do *modo de endereçamento* modificam a interpretação do campo de operando conforme a Tabela 6.

Tabela 6: Modos de endereçamento nas instruções da Pipoca

00 – *Endereçamento imediato*. O valor codificado no campo operando deve ser usado diretamente para a operação definida pela instrução.
01 – *Endereçamento direto*. O valor a ser usado na operação definida pela instrução é o conteúdo da posição de memória cujo endereço está no campo operando da instrução.
10 – *Endereçamento indireto*. O valor a ser usado na operação definida pela instrução é o conteúdo da posição de memória cujo endereço está na posição de memória cujo endereço está no campo operando da instrução.
11 – Este código para o modo de endereçamento não é utilizado na Pipoca.

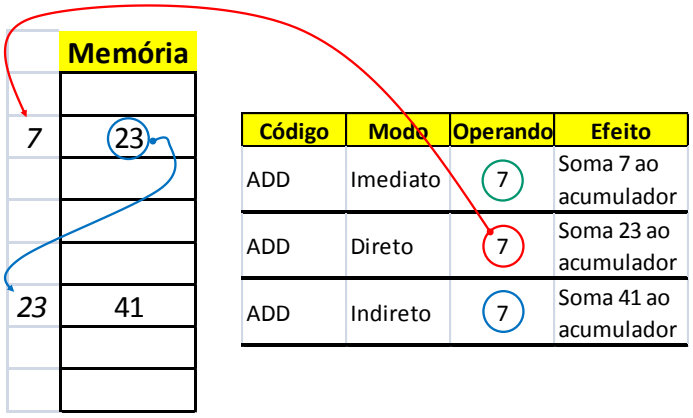


Figura 445: Modos de endereçamento

Supondo que em um dado instante a posição 7 da memória contenha o valor 23, e que a posição 23 da memória contenha o valor 41, uma instrução ADD com o valor 7 codificado em seu campo de operando terá como efeito:

- Se o endereçamento for imediato, somar 7 ao conteúdo do acumulador;
- Se o endereçamento for direto, somar o conteúdo da posição 7 da memória ao registrador – no caso, somar 23 ao acumulador;
- Se o endereçamento for indireto, somar o conteúdo da posição dada pelo conteúdo da posição 7 ao acumulador – no caso, somar 41 ao acumulador.

2 Programação em Assembler

Um programa executável é um mapa da memória, contendo instruções e dados codificados em binário. Entretanto, nenhum ser humano com saúde mental consegue fazer um programa diretamente em binário. O processo de programação em Assembler consiste em:

- preencher uma tabela com os mnemônicos das instruções, dando nomes a posições de memória e usando esses nomes como operandos, sendo assim mais compreensível para humanos, e
- depois, substituir esses mnemônicos e nomes de posições de memória pelos códigos binários correspondentes, um processo a que damos o nome de *montagem*.

O programa em binário deve ser gravado em alguma mídia – hoje em dia, um arquivo, antigamente, fitas de papel ou cartões perfurados – e, no momento da execução, ser carregado na memória do computador. Na Pipoca, simulada no Logisim, isso equivale à carga

de uma imagem na memória principal, o que é possível fazer clicando com o botão direito do mouse sobre o componente memória. Em computadores reais são programas chamados *loaders* que se incumbem de ler mídias com programas binários e carregá-los na memória.

Um mesmo programa tem portanto três formas:

- *Programa Fonte*, que é um texto em formato tabular, escrito e legível por humanos;
- *Programa Executável*, que resulta da montagem do programa fonte, e é normalmente um arquivo com uma imagem da memória. No Logisim, um programa executável é um arquivo ASCII onde cada linha representa uma palavra da memória, codificada em hexadecimal, e
- *Programa em Execução*, que é um conjunto de palavras na memória principal.

A Tabela 7 mostra o código fonte do programa SumX, cuja funcionalidade consiste em somar os valores na memória entre o endereço X e o endereço XEND. Nós vemos ali que o código das instruções (do endereço 0 ao endereço 14) está junto com os dados (do endereço 15 ao 21).

Esses dados consistem no vetor X, que ocupa as posições de 15 a 19, e em duas outras posições: SUM, que irá conter a soma desejada, e P, que será utilizado para endereçar a parcela de X que é somada em cada passo da execução do programa. Na Pipoca, o programa começa a ser executado pela instrução na posição 0 da memória.

O programa SumX tem:

- Uma etapa de inicialização das variáveis (posições de memória) SUM e P, formada pelas instruções nos endereços de 0 a 3, e que atribui o valor inicial 0 para SUM, e coloca em P o endereço X.
- Um *loop* composto pelas instruções colocadas entre o endereço 4 e o endereço 12. Em cada passo deste loop uma das parcelas é adicionada a SUM, e o endereço armazenado em P é incrementado. Ao fim do passo P é comparado com XEND e, dependendo do resultado, o loop é repetido ou o programa passa para a sua fase final.
- Uma etapa de finalização, onde o resultado da soma é encaminhado para a saída e o programa pára.

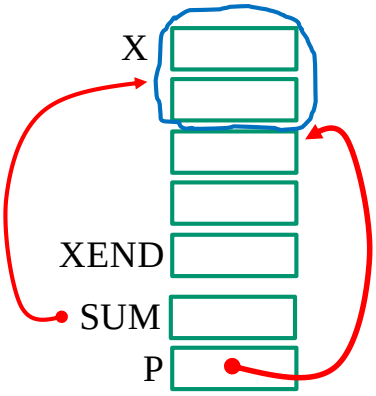


Figura 446: Uso das variáveis SUM e P no programa SumX.

As quatro últimas colunas na Tabela 7 resultam de um processo de montagem que descrevemos a seguir.

Tabela 7: O programa SumX

Label	Size	Address10	Address16	Instruction	Mode	Operand	Comentários	OpCode10	Operand10	Word10	Word16
	1	0	00	LOAD	0	0	Zera o acumulador	8	0	32768	8000
	1	1	01	STORE	0	SUM	Coloca 0 em SUM	10	20	40980	A014
	1	2	02	LOAD	0	X	Carrega o endereço X no acumulador	8	15	32783	800F
	1	3	03	STORE	0	P	Coloca o endereço X em P	10	21	40981	A015
LOOP	1	4	04	LOAD	1	SUM	Carrega o conteúdo de SUM no acumulador	8	20	33812	8414
	1	5	05	ADD	2	P	Soma o conteúdo da posição de memória cujo endereço é P ao acumulador	0	21	2069	0815
	1	6	06	STORE	0	SUM	Coloca o resultado na posição SUM	10	20	40980	A014
	1	7	07	LOAD	1	P	Carrega o conteúdo de P	8	21	33813	8415
	1	8	08	ADD	0	1	Soma 1	0	1	1	0001
	1	9	09	STORE	0	P	Coloca o resultado em P	10	21	40981	A015
	1	10	0A	COMPARE	0	XEND	Compara XEND com o acumulador	1	19	4115	1013
	1	11	0B	JMPLT	0	FINISH	Se for menor, desvia para FINISH	7	13	28685	700D
	1	12	0C	JMP	0	LOOP	Senão, volta para LOOP	4	4	16388	4004
FINISH	1	13	0D	OUTPUT	1	SUM	Coloca o resultado na saída	9	20	37908	9414
	1	14	0E	HALT			Para.	2	0	8192	2000
X	1	15	0F			3142	Números a serem somados	0	3142	3142	0C46
	1	16	10			4542		0	4542	4542	11BE
	1	17	11			3325		0	3325	3325	0CFD
	1	18	12			1234		0	1234	1234	04D2
XEND	1	19	13			8786		0	8786	8786	2252
SUM	1	20	14			0		0	0	0	0000
P	1	21	15			0		0	0	0	0000

3 Montagem do Programa Executável

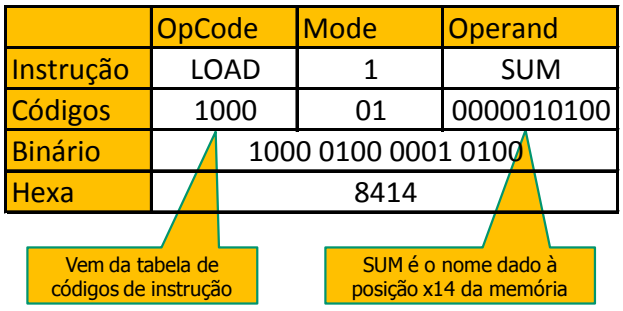


Figura 447: Montagem de uma instrução

A conversão para binário de uma linha de código como LOAD 1 SUM se faz pelas etapas abaixo, ilustradas na Figura 447:

- Na Tabela 5 vemos que o código da instrução LOAD é 1000.
- O modo de endereçamento codificado em binário com 2 bits é 01.
- Olhando o programa na Tabela 7, nós vemos que SUM é o nome (label) dado à posição 20 de memória, que, em binário de 10 bits, é 0000010100.
- A conversão completa resulta da concatenação (justaposição) desses três binários, resultando em 1000010000010100 ou, em hexadecimal, 8414.

Como você pode ter percebido, a tradução da tabela-programa para binário é uma tarefa insana. Mas essa tarefa só foi feita manualmente pelos pioneiros da computação com , pelos parâmetros atuais, enormes custos de verificação e baixíssima produtividade dos programadores. Mas cedo se percebeu que computadores eram muito bons para ... auxiliar a programação de computadores! Um *montador* (um *assembler*) é um programa que lê uma tabela-programa, e gera imagens binárias a serem carregadas nas memórias, automatizando o processo ilustrado na Figura 447.

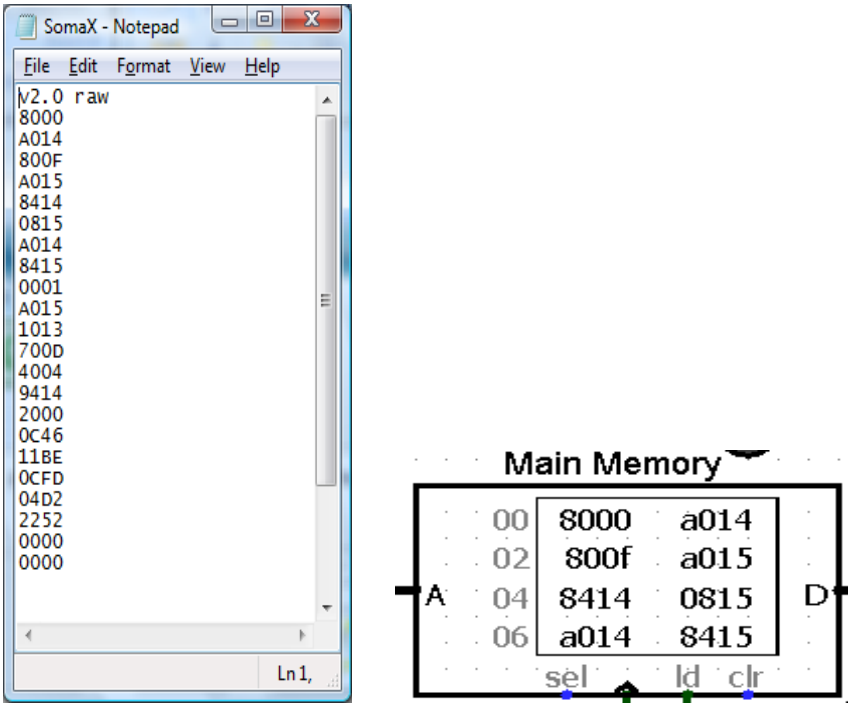


Figura 448: O programa SumX, como um arquivo de imagem de memória do Logisim e carregado na memória

A Pipoca é um circuito simulado no Logisim, onde mapas de memória podem ser carregados clicando sobre a memória com o botão direito do mouse e escolhendo a opção *Load Image*, que abre um diálogo para escolha de um arquivo como o mostrado na Figura 448. Arquivos como esse podem ser produzidos usando a planilha Pipoca.xls, disponível no site do curso, e aqui utilizada como um assembler.

4 O Circuito Principal da CPU Pipoca

Já temos agora condições de apresentar o circuito principal da CPU Pipoca, mostrado na Figura 449.

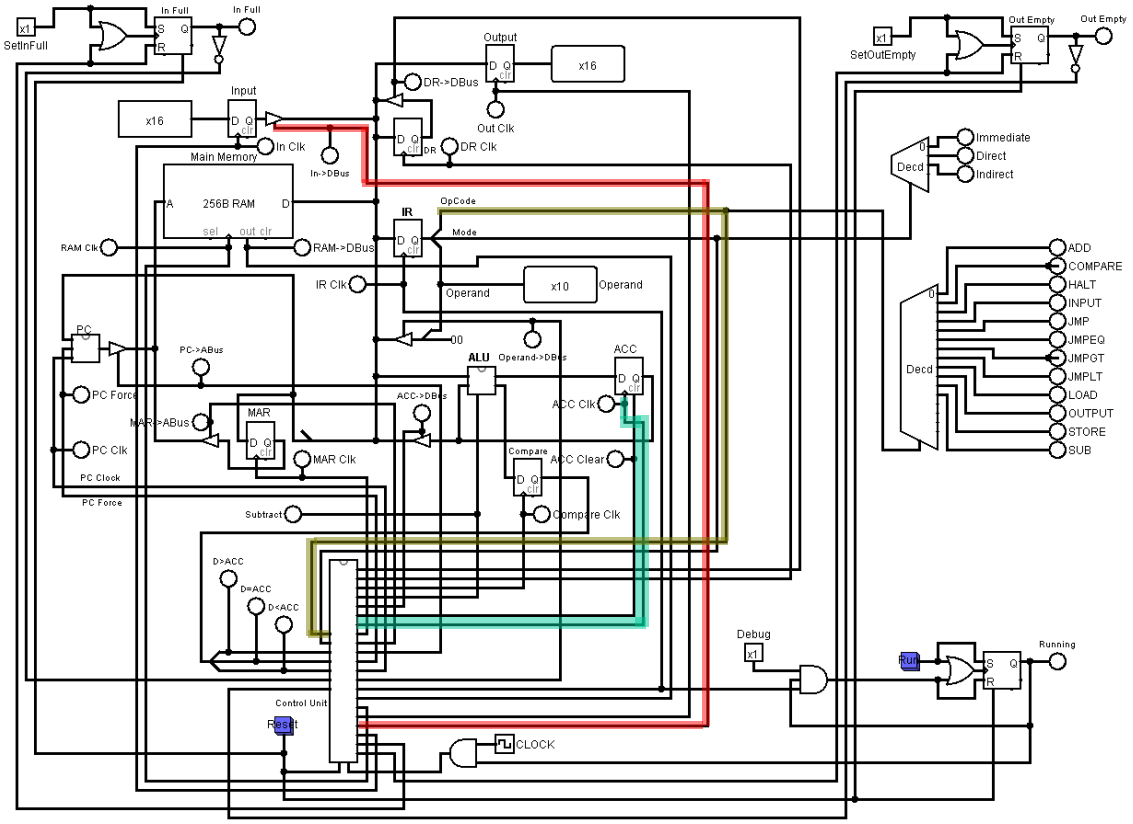


Figura 449: O circuito principal da CPU Pipoca

Complicado? Sem dúvida, mas vamos por partes; mingau quente se come pelas beiradas. Primeiramente, você deve reparar que neste circuito o principal complicador é a *unidade de controle*, o retângulo vertical na parte inferior do circuito. Para ali vão muitos fios, e dali saem outros tantos. Isto não é de se estranhar, posto que a função da unidade de controle é, como dissemos, levantar os sinais de controle na sequëncia e tempos adequados para a implantação dos fluxos de dados que implementam as instruções de máquina. Como exemplos, estão destacados na Figura 449 cabeamentos para o sinal de clock do acumulador (em azul), para a tomada do barramento de dados pelo registrador Input (em vermelho) e para o campo operando da instrução (em marrom). O conjunto completo de entradas e saídas da unidade de controle está mostrado na Figura 450.

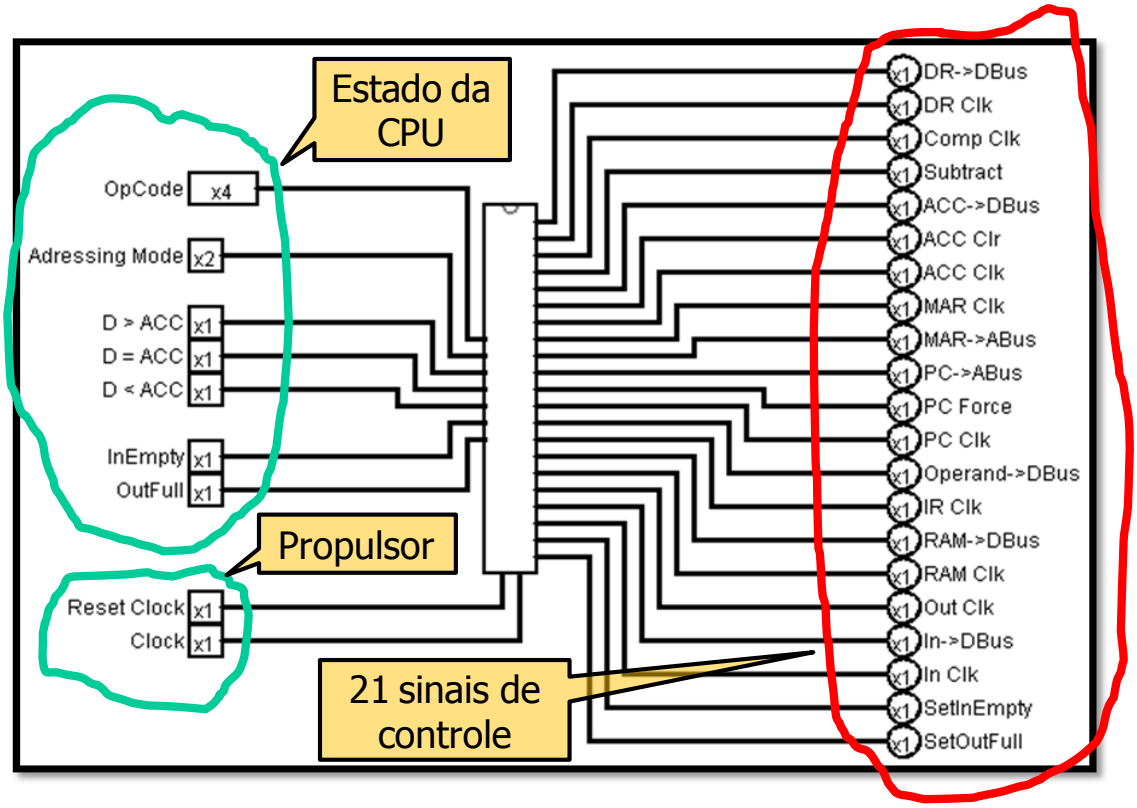


Figura 450: Entradas e saídas da unidade de controle

O circuito principal da CPU Pipoca tem ainda circuitos auxiliares para sincronização de entrada e saída, colocados na parte superior do diagrama, e um circuito para depuração (*debug*) de programas, no canto inferior direito que, bloqueando o sinal de clock, permite que a execução de um programa se interrompa ao término de cada instrução executada.

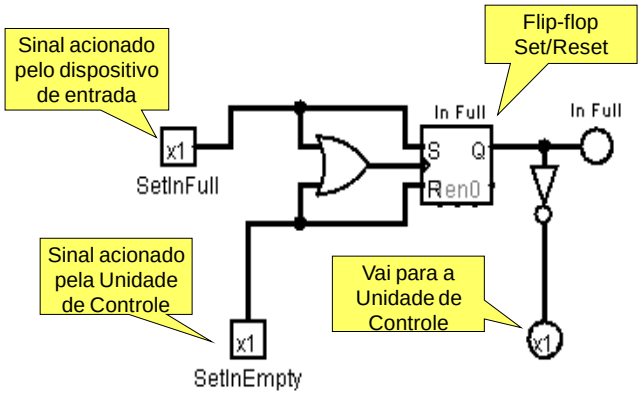


Figura 451: Circuito para sincronização da CPU com dispositivo de entrada

O circuito mostrado na Figura 451 realiza a sincronização da CPU com o (único, na Pipoca) dispositivo de entrada. A sincronização é necessária porque, com a execução automática e incessante de instruções pela CPU, existe tanto a possibilidade de captura de dados nos pinos de entrada antes que estejam prontos, como a de captura repetida de um mesmo dado de entrada, como ainda a de mudança de dados pelo dispositivo de entrada antes de serem lidos pela CPU. O protocolo seguido pela CPU e pelo dispositivo prescreve as seguintes regras:

- Somente o dispositivo de entrada aciona o sinal SetInFull, e somente a CPU aciona o sinal SetInEmpty;
- O dispositivo de entrada somente aciona SetInFull quando o dado de entrada está pronto e InFull = 0, e a CPU (através da Unidade de Controle) somente aciona

SetInEmpty quando InFull = 1 (ou InEmpty = 0), o que é feito após a leitura dos dados de entrada;

- O circuito é inicializado com InFull = 0.

O circuito de sincronização com o dispositivo de saída é similar a este.

Na Pipoca, as instruções INPUT e OUTPUT exigem intervenção do usuário, que deve usar a ferramenta de manipulação do Logisim para escolher valores de entrada e também para sincronizar "dispositivos" externos, apertando convenientemente os botões *SetInFull*(para avisar que um valor de entrada está pronto para ser lido) e *SetOutEmpty*(para avisar que um valor colocado anteriormente no registrador Output já foi consumido).

Programas podem ser executados instrução por instrução, o que é muito útil para a depuração (*debug*) de erros. Para isto, é preciso colocar o valor 1 na entrada Debug, e apertar o botão Run a cada vez que uma nova instrução é carregada no registrador de instrução.

Se retirarmos a Unidade de Controle da CPU Pipoca, retornando à emissão manual dos sinais de controle, e se retirarmos também os circuitos de sincronização de entrada e saída, teremos um circuito como o da Figura 452. Comparando com o circuito da Figura 135 você pode observar que:

- foi acrescentado um *registrador de instruções*, o IR (*Instruction Register*) que tem a função de armazenar a instrução em execução num dado instante;
- temos um barramento de endereços, o ABus, além do barramento de dados DBus;
- com a entrada ligada ao barramento de dados , e com a saída ligada ao barramento de endereços, nós vemos o registrador PC (*Program Counter*), que armazena o endereço da próxima instrução a ser executada; nós veremos adiante a lógica de funcionamento do PC;
- temos também conjuntos de leds ligados a decodificadores que não têm função na lógica do circuito, mas nos ajudam a visualizar qual instrução está em execução e qual o modo de endereçamento empregado na instrução corrente.

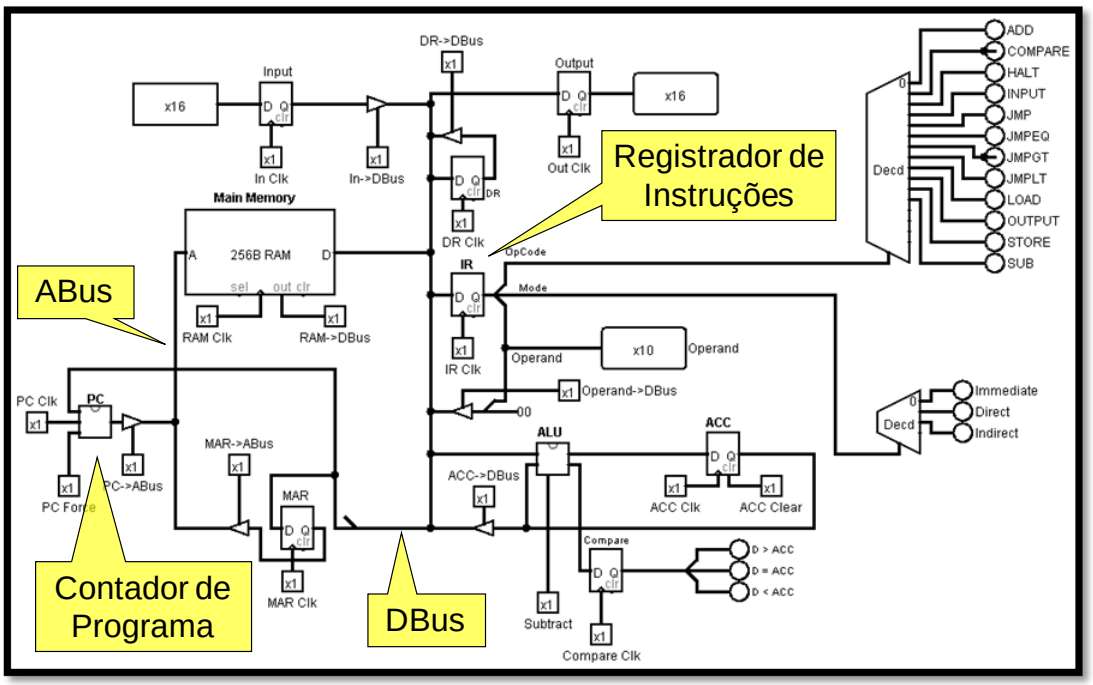


Figura 452: Rotas de dados na CPU Pipoca

5 O Contador de Programa

Outro subcircuito da CPU Pipoca é o contador de programa, mostrado na Figura 453.

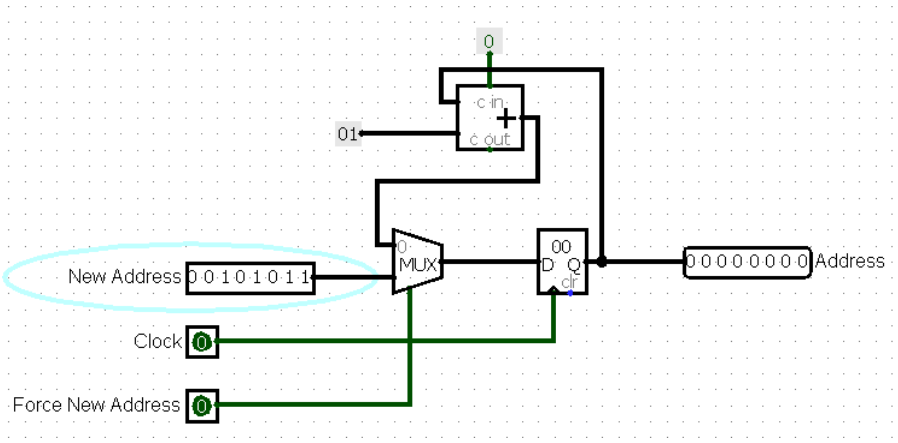


Figura 453: O contador de programa

Nesse circuito observamos que:

- o sinal Clock faz com que o registrador copie a sua entrada, que é alimentada pela saída de um multiplexador;
- este multiplexador escolhe, segundo a entrada Force New Address, entre o valor corrente do registrador acrescido de 1 (resultado da operação de soma) e o valor constante na entrada New Address.

A unidade de controle se encarrega de colocar nessas entradas os valores adequados nos tempos corretos.

6 A Unidade de Controle

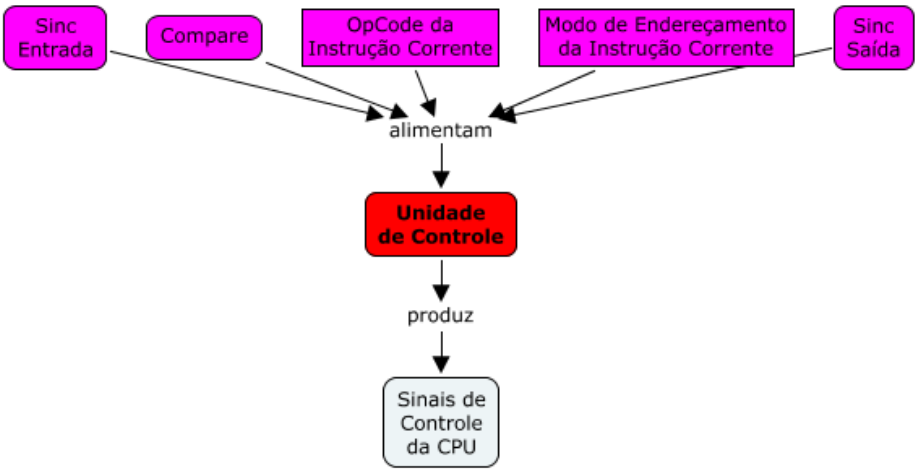


Figura 454: O papel da Unidade de Controle

A Figura 454 ilustra a função da unidade de controle, que pode ser vista com mais detalhes no circuito da Figura 450. Para cumprir este papel a unidade de controle da Pipoca possui como principais componentes:

- um circuito *Timing* que fornece os sinais de tempo que conduzem a seqüência de eventos na unidade de controle,
- uma memória ROM (*Read Only Memory*) que contém uma tabela de desvios, a *Branch Table*, cuja função explicaremos a seguir,

- uma outra ROM que abriga o microprograma, e
- registradores da unidade de controle.

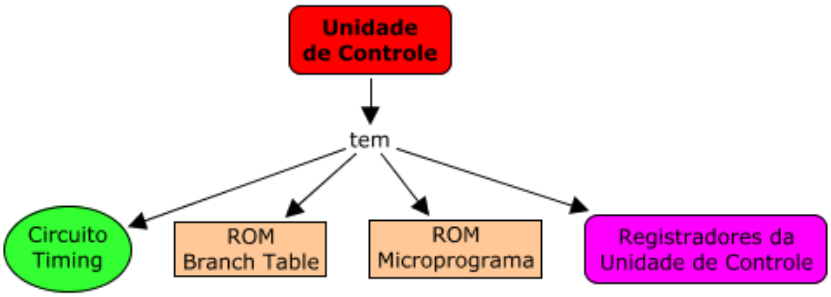


Figura 455: Componentes da Unidade de Controle

Os registradores da unidade de controle da Pipoca são:

- o *mPC* (micro Program Counter), que contém o endereço da micro-instrução em execução,
- o *mIR* (micro Instruction Register), que armazena a micro-instrução em execução,
- o registrador *Signals*, cuja saída é também a saída da unidade de controle, e que fornece os sinais de controle para a CPU,
- e o *Return Address*, que é usado para permitir o uso de sub-rotinas, isto é, de sequências de micro-instruções que são reaproveitadas, como veremos a seguir.

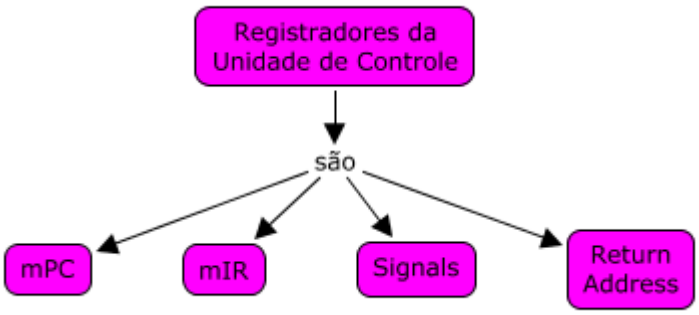


Figura 456: Registradores da Unidade de Controle

A unidade de controle utiliza uma técnica conhecida como *microprogramação* para produzir os sinais de controle que, emitidos nos momentos adequados para barramentos, registradores e memória, executam efetivamente os fluxos de dados que correspondem às instruções.

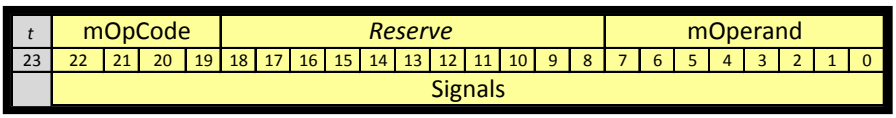


Figura 457: Formato de uma microinstrução

Em uma CPU microprogramada uma instrução é executada como uma sequência de *microinstruções*. Microinstruções da Pipoca têm 24 bits, no formato mostrado na Figura 457. Temos dois tipos de microinstruções:

- *microinstruções de sinal*, onde os bits de 0 a 22 são valores para os sinais de controle (clocks de registradores e memórias, controles de barramentos) que a unidade de controle deve prover para o circuito principal da CPU, ou

- *microinstruções de desvio*, que são utilizadas para guiar o fluxo de execução das microinstruções.

O tipo da microinstrução é ditado pelo bit mais significativo, o bit 23.

Tabela 8: Microinstruções de Desvio

m-Instruction	DEC	BIN	Effect
jmpNext	0	0000	Desvia para a micro-instrução inicial da instrução no registrador IR da CPU
jmpIMMEDIATE	1	0001	Desvia para o operando se o modo de endereçamento for Imediato
jmpDIRECT	2	0010	Desvia para o operando se o modo de endereçamento for Direto
jmpINDIRECT	3	0011	Desvia para o operando se o modo de endereçamento for Indireto
error	4	0100	Situação inesperada - não deveria acontecer. Acende um led.
return	5	0101	Desvia para a micro-instrução apontada pelo registrador Return Addr
jsr	6	0110	Desvia para o operando e armazena o endereço consecutivo no Return Addr
jmpEQ	7	0111	Desvia para o operando se D = ACC for igual a 1
jmpGT	8	1000	Desvia para o operando se D > ACC for igual a 1
jmpLT	9	1001	Desvia para o operando se D < ACC for igual a 1
jmp	10	1010	Desvia para o operando incondicionalmente
jmpInEmpty	11	1011	Desvia para o operando se InEmpty = 1
jmpOutFull	12	1100	Desvia para o operando se OutFull = 1
	13	1101	*Estes códigos podem ser usados em novas microinstruções*
	14	1110	
	15	1111	

Instruções podem ter etapas comuns em sua execução, como a obtenção do operando conforme o modo de endereçamento. O micro-código destas etapas é reaproveitado usando a micro-instruções jsr (*jump to subroutine*), que desvia para o endereço dado por seu operando, e armazena o endereço atual acrescido de 1 no registrador Return Address, e a micro-instrução return, que desvia para o endereço armazenado no registrador Return Address.

O circuito da unidade de controle pode ser visto na Figura 458, onde se pode destacar:

- as entradas
  - *Opcode*, com o código da instrução corrente,
  - *Mode*, com o modo de endereçamento,
  - *In Empty*, *Out Full*, com o estado dos registradores de sincronização de entrada e saída,

    - *D>ACC*, *D=ACC*, *D<ACC*, com o resultado da última instrução COMPARE executada, e
    - *Reset Clock* e *CLOCK*, que servem para inicializar o circuito Timing e para dar vida à CPU;
- as saídas com sinais que controlam o fluxo de dados na CPU;
- a memória ROM *Microprogram*, que abriga o microprograma, com 256 palavras de 24 bits;
- a memória ROM *Branch Table*, com 16 palavras de 8 bits, que armazena para cada código de instrução (*Opcode*) o endereço da primeira microinstrução a ser executada para a execução da instrução;
- o registrador *mIR* (micro Instruction Register), com 24 bits, que armazena a micro-instrução corrente;
- o circuito *mPC* (micro Program Counter), que funciona de forma análoga ao contador de programa da CPU;
- o registrador *Return Addr*, de 8 bits, que armazena o endereço de retorno para uma micro-instrução jsr;
- o circuito *Timing*, inicializado pela entrada *Reset Clock* e alimentado pelo *CLOCK*, que ciclicamente oferece os sinais t0, t1 e t2;
- o registrador *Signals*, que armazena os sinais utilizados no controle do fluxo de dados da CPU;
- um *splitter* ligando o registrador *Signals* aos pinos de saída;
- um decodificador de código de instrução;
- um decodificador de modo de endereçamento;
- portas lógicas que essencialmente implementam decisões de desvio no fluxo de micro-instruções;
- e alguns leds que animam a festa.

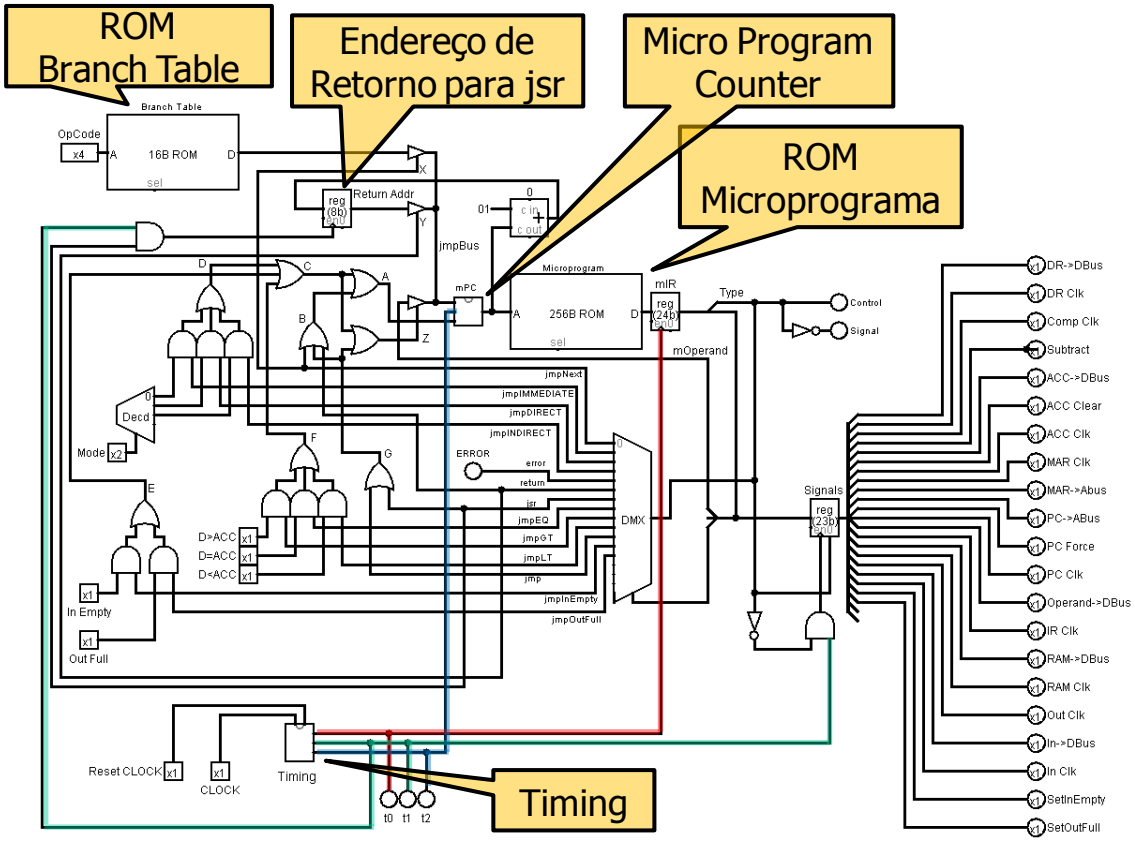


Figura 458: A unidade de controle

7 O Ciclo de Micro-Instrução

Um ciclo de instrução é na verdade implementado por diversos ciclos de micro-instrução, que, de forma similar aos ciclos de instrução, são compostos por etapas de leitura e de execução de micro-instruções que se alternam indefinidamente.

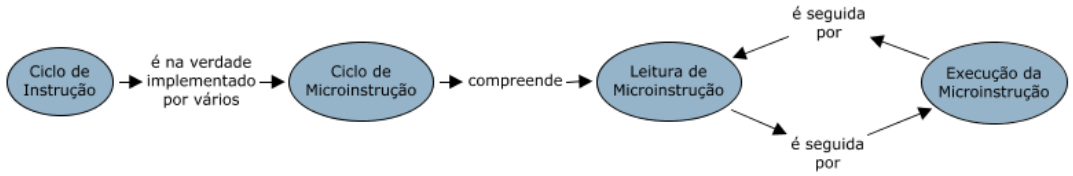


Figura 459: Ciclo de micro-instrução

Na etapa de leitura, uma micro-instrução é lida na ROM de microprograma, no endereço apontado pelo registrador mPC, e a instrução lida é armazenada no registrador mIR.

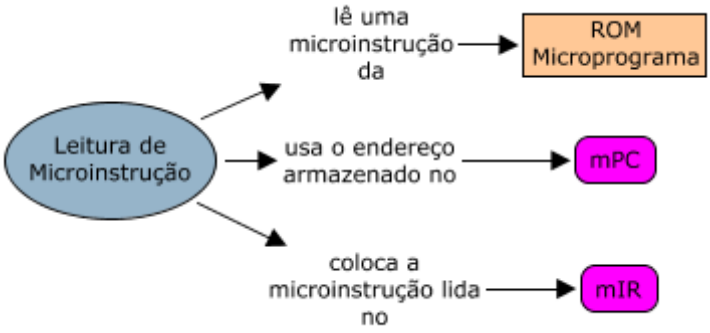


Figura 460: Leitura de uma micro-instrução

A execução da micro-instrução pode alterar os valores dos registradores da unidade de controle e, em particular, do registrador Signals que, como vimos, emite os sinais de controle para a CPU, e do registrador mPC, determinando assim a próxima micro-instrução a ser executada.

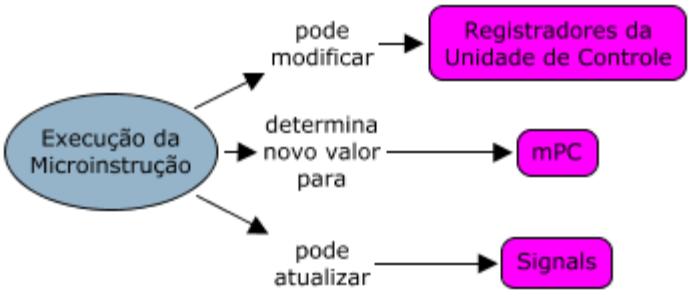


Figura 461: Execução de uma micro-instrução

O mIR tem a interpretação de sua saída dependente do tipo da micro-instrução.

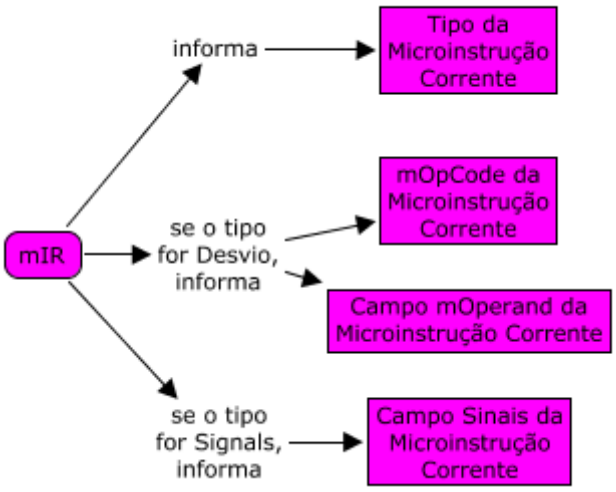

informa
Tipo da Microinstrução Corrente
mOpCode da Microinstrução Corrente
mIR
se o tipo for Desvio, informa
Campo mOperand da Microinstrução Corrente
se o tipo for Signals, informa
Campo Sinais da Microinstrução Corrente

Figura 462: Saída do registrador mIR

O ciclo de micro-instrução é comandado pelos sinais emitidos pelo circuito Timing.

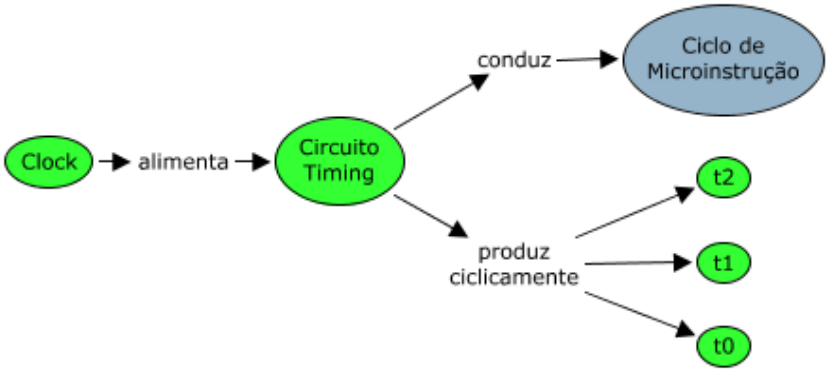


Figura 463: Sinais do ciclo de micro-instrução

O circuito Timing está mostrado na Figura 464, sendo similar ao circuito da Figura 139.

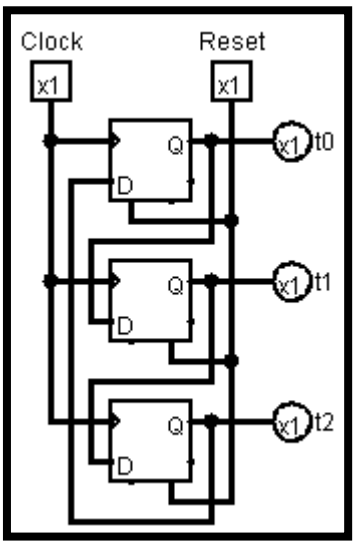


Figura 464: O circuito Timing

As observações abaixo podem ajudar a compreender o funcionamento da unidade de controle:

- O circuito *Timing* gera ciclicamente os sinais t0, t1 e t2;
- Quanto t0 passa de 0 para 1 (veja o destaque em vermelho na Figura 458), a microinstrução no endereço apontado por *mPC* é carregada no *mIR*;

Figura 465: Ações em t0

- Quando t1 passa de 0 para 1 (destaque em verde na Figura 458), se a microinstrução for do tipo *signal*, seu operando é carregado no registrador *Signals*, cujas saídas fornecem os sinais de controle para a *CPU*; senão, se a microinstrução for *jsr* (jump subroutine), o valor do *mPC* acrescido de 1 é armazenado no registrador *Return Address*

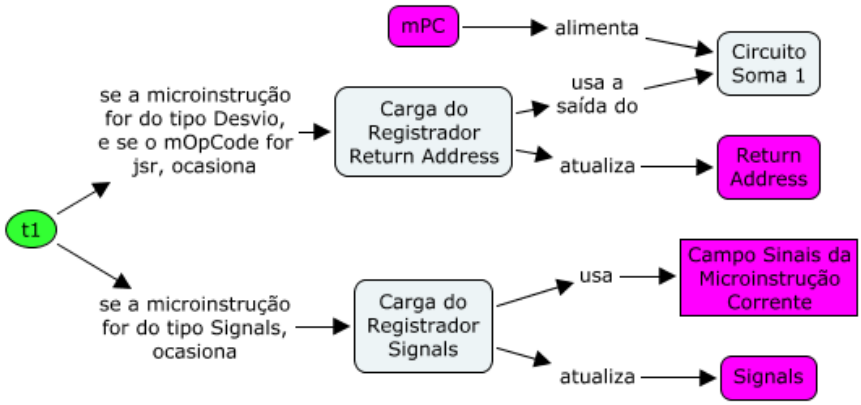


Figura 466: Ações em t1

- Quando t2 passa de 0 para 1 (destaque em azul na Figura 458), o clock do *mPC* é acionado, atualizando seu conteúdo, o que determina a próxima micro-instrução a ser executada. O novo conteúdo depende do sinal aplicado à entrada *Force New Address* do *mPC*:
  - se for igual a 0, será o endereço consecutivo ao conteúdo anterior;
  - se for igual a 1, será o conteúdo da entrada *New Address* do *mPC*;

  O circuito que decide o valor aplicado à entrada *Force New Address* é um OR de várias cláusulas:
  - a microinstrução corrente é *jmp*, *jmpNext*, ou *return*, ou *jsr*;
  - a microinstrução corrente é *jmpImmediate* e o modo de endereçamento é *Immediate*, ou a microinstrução é *jmpDirect* e o modo de endereçamento é *Direct*, ou a microinstrução é *jmpIndirect* e o modo de endereçamento é *Indirect*;
  - a microinstrução corrente é *jmpGT* e *D>ACC* = 1, ou *jmpEQ* e *D=ACC* = 1, ou *jmpLT* e *D<ADD* = 1;
  - a microinstrução corrente é *jmpInFull* e *InFull* = 1, ou *jmpOutEmpty* e *OutEmpty* = 1.

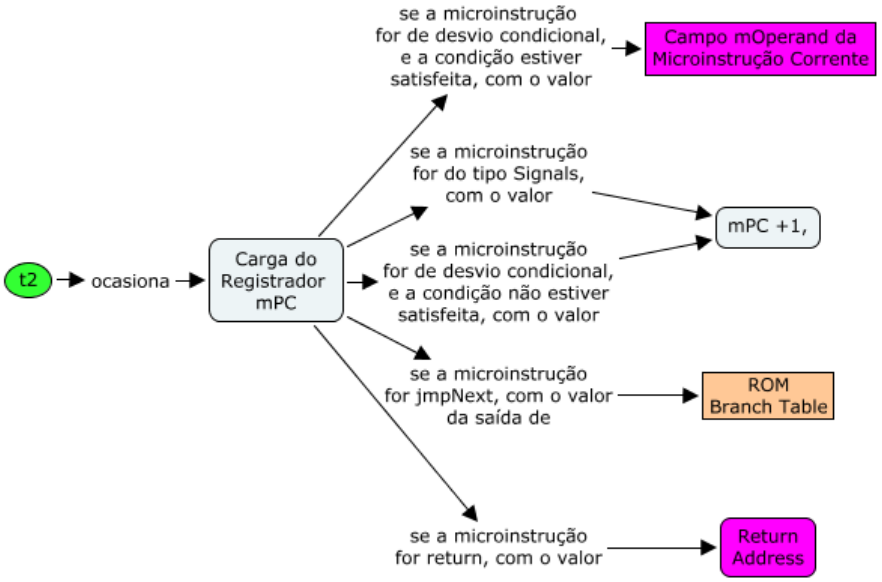

se a microinstrução for de desvio condicional, e a condição estiver satisfeita, com o valor
Campo mOperand da Microinstrução Corrente
se a microinstrução for do tipo Signals, com o valor
mPC +1,
t2
ocasiona
Carga do Registrador mPC
se a microinstrução for de desvio condicional, e a condição não estiver satisfeita, com o valor
se a microinstrução for jmpNext, com o valor da saída de
ROM Branch Table
se a microinstrução for return, com o valor
Return Address

Figura 467: Ações em t2

Quanto ao conteúdo da entrada *New Address* do *mPC*, isto é, quanto ao endereço da próxima microinstrução a ser executada no caso de desvio, ele será:

- o conteúdo da memória *Branch Table* no endereço dado por OpCode se a microinstrução corrente for *jmpNext*,

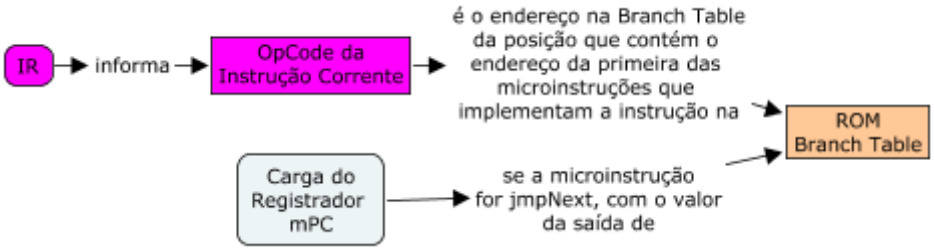

IR
informa
OpCode da Instrução Corrente
é o endereço na Branch Table da posição que contém o endereço da primeira das microinstruções que implementam a instrução na
ROM Branch Table
Carga do Registrador mPC
se a microinstrução for jmpNext, com o valor da saída de

Figura 468: Efeito da micro-instrução jmpnext

- o conteúdo do registrador *Return Address* se a microinstrução corrente for *return*,
- o conteúdo do campo operando da microinstrução se esta for *jmp* ou *jsr*, ou se a microinstrução for *jmp*<condição> e <condição> for igual a 1.

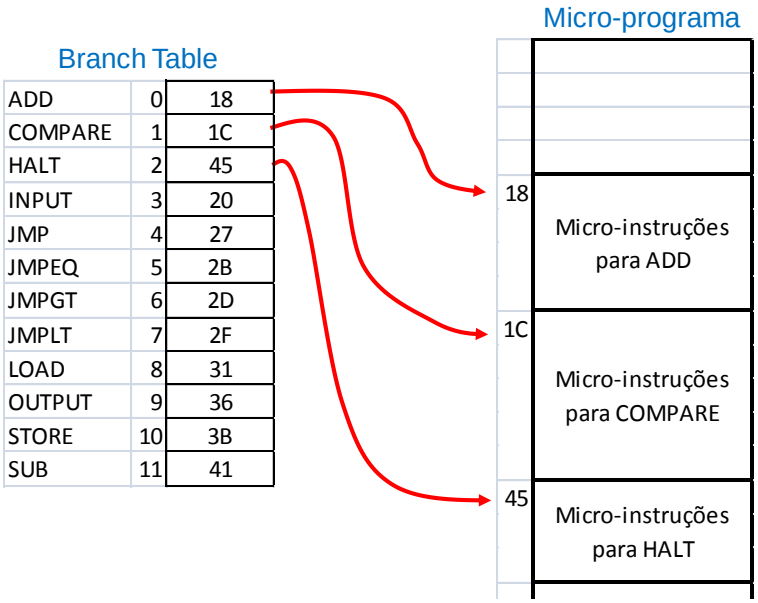


Figura 469: A Branch Table e o micro-programa

A Figura 469 mostra o relacionamento entre a Branch Table e o micro-programa.

8 O Micro-Programa

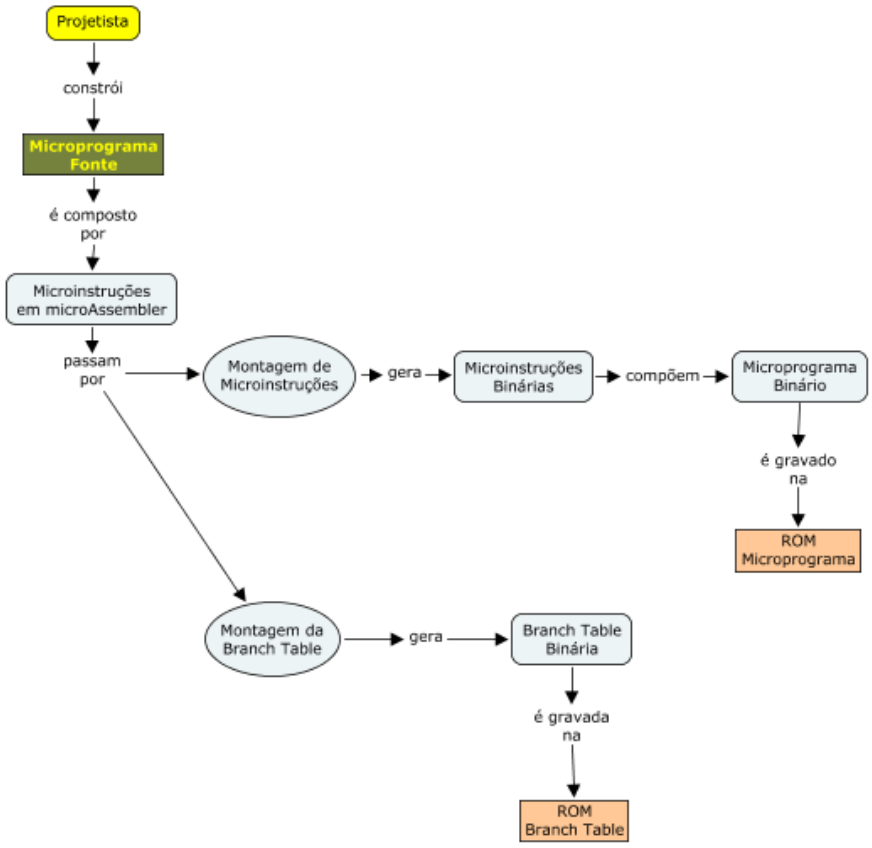


Figura 470: Construção do micro-programa e da Branch Table

Micro-programas também são feitos por humanos, os projetistas do computador. O processo de micro-programação consiste no preenchimento de uma tabela utilizando uma linguagem comumente chamada micro-assembler. Nesta linguagem utilizam-se labels para se referir a posições na memória de micro-programa, mnemônicos para as micro-instruções de desvio, e indicações explícitas de quais sinais devem ser ativados para as micro-instruções de sinal.

A tabela obtida é o micro-programa fonte, legível por humanos, a partir do qual processos de montagem produzem o micro-programa binário e também a Branch Table.

Addr(HEX)	Address	Label	m-Instruction	mOperand	t	mOpCode				*Reserve*								mOperand								Comments
					23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
					Type	*New Signal*	*New Signal*	SetOutFull	SetInEmpty	In Clk	In->DBus	Out Clk	RAM Clk	RAM->DBus	IR Clk	Operand->DBus	PC Clk	PC Force	PC->ABus	MAR->Abus	MAR Clk	ACC Clk	ACC Clear	ACC->Dbus	Subtract	Comp Clk	DR Clk	DR->Dbus	
						Signals																							
00	0	Fetch			0														1									Saída do PC vai para Abus
01	1				0									1					1									Saída da RAM vai para Dbus
02	2				0									1	1				1									IR copia o Dbus
03	3				0												1											PC avança
04	4		jmpNext		1																							Inicia a execução da instrução armazenada no IR

Figura 471: Micro-código para fetch de instrução

A Figura 471 mostra o micro-código que implementa a leitura (*fetch*) de uma nova instrução. Este micro-código é executado ao fim de cada instrução; todas as instruções sempre terminam pela micro-instrução *jmp Fetch*. Repare que a última micro-instrução de Fetch é *jmpNext* que, como vimos, desvia para a posição apontada pela saída da ROM Branch Table.

Addr(HEX)	Address	Label	m-Instruction	mOperand	t	mOpCode				Reserve										mOperand								Comments
					23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
						Signals																							
					Type	New Signal	New Signal	SetOutFull	SetInEmpty	In Clk	In->DBus	Out Clk	RAM Clk	RAM->DBus	IR Clk	Operand->DBus	PC Clk	PC Force	PC->ABus	MAR->Abus	MAR Clk	ACC Clk	ACC Clear	ACC->Dbus	Subtract	Comp Clk	DR Clk	DR->Dbus	
18	24	ADD	jsr	GetOperand	1																							Resolve endereçamento; operando efetivo no DR
19	25				0																						1	Saída do DR vai para Dbus
1A	26				0																	1					1	ACC copia saída da ALU
1B	27		jmp	Fetch	1																							Fetch da próxima instrução
41	65	SUB	jsr	GetOperand	1																							Resolve endereçamento; operando efetivo no DR
42	66				0																				1		1	Saída do DR vai para Dbus; pino Subtract da ALU = 1
43	67				0																	1			1		1	ACC copia a saída da ALU (=ACC - Dbus)
44	68		jmp	Fetch	1																							Fetch da próxima instrução

Figura 472: Micro-código para as instruções ADD e SUB

Na Figura 472 nós vemos o micro-código das micro-instruções ADD e SUB. Ele se inicia com a micro-instrução *jsr GetOperand*; o micro-código de *GetOperand* (Figura 473) se encarrega de resolver o modo de endereçamento, colocando o operando efetivo da instrução no registrador DR.

Addr(HEX)	Address	Label	m-Instruction	mOperand	t	mOpCode				Reserve										mOperand								Comments
					23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
						Signals																							
					Type	New Signal	New Signal	SetOutFull	SetInEmpty	In Clk	In->DBus	Out Clk	RAM Clk	RAM->DBus	IR Clk	Operand->DBus	PC Clk	PC Force	PC->ABus	MAR->Abus	MAR Clk	ACC Clk	ACC Clear	ACC->Dbus	Subtract	Comp Clk	DR Clk	DR->Dbus	
05	5	GetOperand	jmpIMMEDIATE	Immediate	1																							
06	6		jmpDIRECT	Direct	1																							Desvia para micro-instruções apropriadas
07	7		jmpINDIRECT	Indirect	1																							
08	8		error	GetOperand	1																							Modo de endereçamento inexistente??!!
09	9	Immediate			0											1												O operando da instrução vai para o Dbus
0A	10				0											1										1		DR copia o Dbus
0B	11		return		1																							Retorna
0C	12	Direct			0											1												O operando da instrução vai para o Dbus
0D	13				0											1				1	1							MAR copia o Dbus; saída do MAR vai para o Abus
0E	14				0									1						1								Saída da RAM vai para Dbus
0F	15				0									1						1						1		DR copia o Dbus
10	16		return		1																							Retorna
11	17	Indirect			0											1												O operando da instrução vai para o Dbus
12	18				0											1				1	1							MAR copia o Dbus; saída do MAR vai para o Abus
13	19				0									1						1								Saída da RAM vai para Dbus
14	20				0									1						1	1							MAR copia o Dbus; saída do MAR vai para o Abus
15	21				0									1						1								Saída da RAM vai para Dbus
16	22				0									1						1						1		DR copia o Dbus
17	23		return		1																							Retorna

Figura 473: Micro-código para a subrotina GetOperand

Com exceção da instrução HALT, o micro-código de todas as outras micro-instruções invoca a sub-rotina *GetOperand*. No início da execução de *GetOperand* é feito um desvio para sequências micro-instruções que emitem os sinais de controle necessários para a obtenção do operando segundo o modo de endereçamento. Essas sequências têm os labels *Immediate*, *Direct* e *Indirect*, e todas as três terminam com uma micro-instrução *return*, que retorna para o ponto onde a *GetOperand* foi chamada.

Addr(HEX)	Address	Label	m-Instruction	mOperand	t	mOpCode				Reserve										mOperand								Comments
					23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
						Signals																							
					Type	New Signal	New Signal	SetOutFull	SetInEmpty	In Clk	In->DBus	Out Clk	RAM Clk	RAM->DBus	IR Clk	Operand->DBus	PC Clk	PC Force	PC->ABus	MAR->Abus	MAR Clk	ACC Clk	ACC Clear	ACC->Dbus	Subtract	Comp Clk	DR Clk	DR->Dbus	
27	39	JMP	jsr	GetOperand	1																							Resolve endereçamento; operando efetivo no DR
28	40				0													1									1	Saída do DR vai para Dbus; força desvio no PC
29	41				0												1	1									1	PC copia Dbus
2A	42		jmp	Fetch	1																							Fetch da próxima instrução
2B	43	JMPEQ	jmpEQ	JMP	1																							Desvio condicional para JMP
2C	44		jmp	Fetch	1																							Fetch da próxima instrução
2D	45	JMPGT	jmpGT	JMP	1																							Desvio condicional para JMP
2E	46		jmp	Fetch	1																							Fetch da próxima instrução
2F	47	JMPLT	jmpLT	JMP	1																							Desvio condicional para JMP
30	48		jmp	Fetch	1																							Fetch da próxima instrução

Figura 474: Micro-código para as instruções JMP, JMPEQ, JMPGT e JMPLT

O micro-código para as instruções de desvio está mostrado na Figura 474. Repare que o micro-código das instruções de desvio condicional se inicia pelas micro-instruções de desvio condicional correspondentes. Se a condição de desvio não estiver satisfeita, nada acontece e a micro-instrução *jmp Fetch* é executada para a leitura da próxima instrução.

Addr(HEX)	Address	Label	m-Instruction	mOperand	t	mOpCode				Reserve										mOperand								Comments
					23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
						Signals																							
					Type	*New Signal*	*New Signal*	SetOutFull	SetInEmpty	In Clk	In->DBus	Out Clk	RAM Clk	RAM->DBus	IR Clk	Operand->DBus	PC Clk	PC Force	PC->ABus	MAR->Abus	MAR Clk	ACC Clk	ACC Clear	ACC->Dbus	Subtract	Comp Clk	DR Clk	DR->Dbus	
20	32	INPUT	jsr	GetOperand	1																								Resolve endereçamento; operando efetivo no DR
21	33	InLoop	jmpInEmpty	InLoop	1																								Espera InEmpty = 0
22	34				0																							1	Saída do DR vai para Dbus
23	35				0															1	1							1	MAR copia o Dbus; saída do MAR vai para o Abus
24	36				0					1	1									1									In copia a entrada; saída do In vai para Dbus
25	37				0				1		1		1							1									Zera o FF InEmpty; RAM copia o Dbus
26	38		jmp	Fetch	1																								Fetch da próxima instrução
36	54	OUTPUT	jsr	GetOperand	1																								Resolve endereçamento; operando efetivo no DR
37	55	OutLoop	jmpOutFull	OutLoop	1																								Espera OutFull = 0
38	56				0																							1	Saída do DR vai para Dbus
39	57				0			1				1																1	Zera o FF OutFull; Out copia Dbus
3A	58		jmp	Fetch	1																								Fetch da próxima instrução

Figura 475: Micro-código para as instruções INPUT e OUTPUT

Na Figura 475 nós vemos o micro-código para as instruções INPUT e OUTPUT que, logo após a obtenção do operando, executam um loop de espera composto por uma única micro-instrução, que é executada repetidamente (através de um desvio condicional para ela mesma) até que a condição de sincronização de entrada ou de saída esteja satisfeita.

Addr(HEX)	Address	Label	m-Instruction	mOperand	t	mOpCode				Reserve										mOperand								Comments
					23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
						Signals																							
					Type	*New Signal*	*New Signal*	SetOutFull	SetInEmpty	In Clk	In->DBus	Out Clk	RAM Clk	RAM->DBus	IR Clk	Operand->DBus	PC Clk	PC Force	PC->ABus	MAR->Abus	MAR Clk	ACC Clk	ACC Clear	ACC->Dbus	Subtract	Comp Clk	DR Clk	DR->Dbus	
31	49	LOAD			0																		1						Zera o ACC
32	50		jsr	GetOperand	1																								Resolve endereçamento; operando efetivo no DR
33	51				0																							1	Saída do DR vai para Dbus
34	52				0																	1						1	ACC copia sua entrada (= 0 + Dbus)
35	53		jmp	Fetch	1																								Fetch da próxima instrução
3B	59	STORE	jsr	GetOperand	1																								Resolve endereçamento; operando efetivo no DR
3C	60				0																							1	Saída do DR vai para Dbus
3D	61				0															1	1							1	MAR copia o Dbus; saída do MAR vai para o Abus
3E	62				0															1				1					Saída do ACC vai para Dbus
3F	63				0								1							1				1					RAM copia o Dbus
40	64		jmp	Fetch	1																								Fetch da próxima instrução

Figura 476: Micro-código para as instruções LOAD e STORE

O micro-código para as instruções LOAD e STORE está mostrado na Figura 476, e não apresenta novidades.

Addr(HEX)	Address	Label	m-Instruction	mOperand	t	mOpCode				Reserve										mOperand								Comments
					23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
						Signals																							
					Type	*New Signal*	*New Signal*	SetOutFull	SetInEmpty	In Clk	In->DBus	Out Clk	RAM Clk	RAM->DBus	IR Clk	Operand->DBus	PC Clk	PC Force	PC->ABus	MAR->Abus	MAR Clk	ACC Clk	ACC Clear	ACC->Dbus	Subtract	Comp Clk	DR Clk	DR->Dbus	
45	69	HALT	jmp	HALT	1																								Loop infinito - parada final

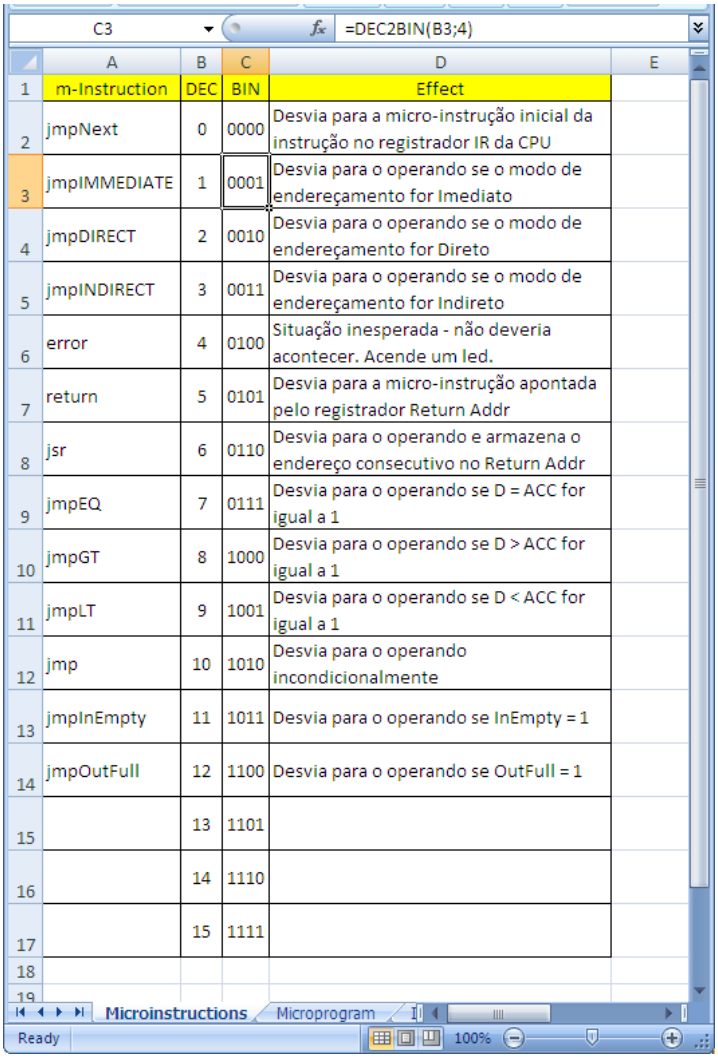
Figura 477: Micro-código para a instrução HALT

A instrução HALT é implantada por uma micro-instrução de desvio incondicional para ela mesma, o que coloca a CPU em um loop infinito, e para o processamento.

9 A Planilha Pipoca.xls

Para funcionar a Pipoca depende de ter preenchidas suas memórias RAM e ROM, o que pode ser feito no Logisim através da carga de arquivos com mapas de memória. Mas como fazer para obter mapas que correspondem a uma intenção do projetista de instruções ou do programador? Sua codificação direta bit por bit é tarefa insana. O arquivo Pipoca.xls serve exatamente para isto, cumprindo as funções de assembler e de micro-assembler, como descrevemos nas próximas seções. Este arquivo contém as planilhas Microinstructions, Microprogram, Instructions, SumX, ReadX, SearchX e Aux.

A planilha Microinstructions pode ser vista na Figura 478, e contém em cada linha um mnemônico da micro-instrução, seu código em decimal e binário, e uma descrição. Ali também pode ser vista a única fórmula utilizada, que utiliza a função Excel DEC2BIN para converter um número decimal para binário.



	A	B	C	D	E
1	m-Instruction	DEC	BIN	Effect	
2	jmpNext	0	0000	Desvia para a micro-instrução inicial da instrução no registrador IR da CPU	
3	jmpIMMEDIATE	1	0001	Desvia para o operando se o modo de endereçamento for Imediato	
4	jmpDIRECT	2	0010	Desvia para o operando se o modo de endereçamento for Direto	
5	jmpINDIRECT	3	0011	Desvia para o operando se o modo de endereçamento for Indireto	
6	error	4	0100	Situação inesperada - não deveria acontecer. Acende um led.	
7	return	5	0101	Desvia para a micro-instrução apontada pelo registrador Return Addr	
8	jsr	6	0110	Desvia para o operando e armazena o endereço consecutivo no Return Addr	
9	jmpEQ	7	0111	Desvia para o operando se D = ACC for igual a 1	
10	jmpGT	8	1000	Desvia para o operando se D > ACC for igual a 1	
11	jmpLT	9	1001	Desvia para o operando se D < ACC for igual a 1	
12	jmp	10	1010	Desvia para o operando incondicionalmente	
13	jmpInEmpty	11	1011	Desvia para o operando se InEmpty = 1	
14	jmpOutFull	12	1100	Desvia para o operando se OutFull = 1	
15		13	1101		
16		14	1110		
17		15	1111		
18					
19					

Figura 478: A planilha Microinstructions

A planilha Microprogram, mostrada na Figura 479 é, de longe, a mais complicada no arquivo Pipoca.xls. As colunas onde o microprograma é definido são aquelas com cabeçalhos de cor palha; as colunas com cabeçalhos de cor cinza são campos calculados. Passamos agora a descrever suas colunas.

Colunas Addr(HEX) e Address. A coluna Address é simplesmente uma enumeração de endereços consecutivos a serem ocupados na memória de microprograma, em decimal; a

coluna Addr(Hex) é obviamente sua conversão para hexadecimal, muito úteis para acompanhamento da execução de um programa no Logisim.

Colunas Label, m-Instruction e Operand. É nessas colunas que a parte de controle do microprograma é definida. Para microinstruções de sinais estes campos devem ser mantidos vazios.

Coluna Type. Essa coluna contém 0 se a microinstrução for de controle, e 1 se for de sinais, valor calculado por uma fórmula que simplesmente verifica se a coluna m-Instruction da mesma linha está vazia.

Colunas Signals. São os sinais que a unidade de controle deve enviar à CPU quando a microinstrução for do tipo Signal. O micro-programador deve colocar 1s nas posições correspondentes aos sinais desejados.

Coluna mOpCode(DEC). É um campo calculado que tem o valor 0 se o campo m-Instruction estiver vazio na linha, e senão, o código decimal da microinstrução obtida por pesquisa por seu mnemônico na planilha Microinstruction.

Coluna Operand(DEC). É um campo calculado que tem o valor 0 se o campo Operand estiver vazio na linha, e senão, o valor da coluna Address na linha onde o valor na Coluna Label éigual ao campo Operand.

Colunas Bin2Dec Conversion. Estas colunas fazem a conversão para decimal de cada bit nas colunas Type e Signals, multiplicando o bit pela potência de 2 correspondente à sua posição na microinstrução.

Coluna Microinstruction Word(DEC). Esta coluna contém o valor em decimal da palavra de 24 bits contendo a microinstrução. Para compreender sua fórmula de cálculo é preciso examinar o formato das microinstruções, mostrado na Figura 457.

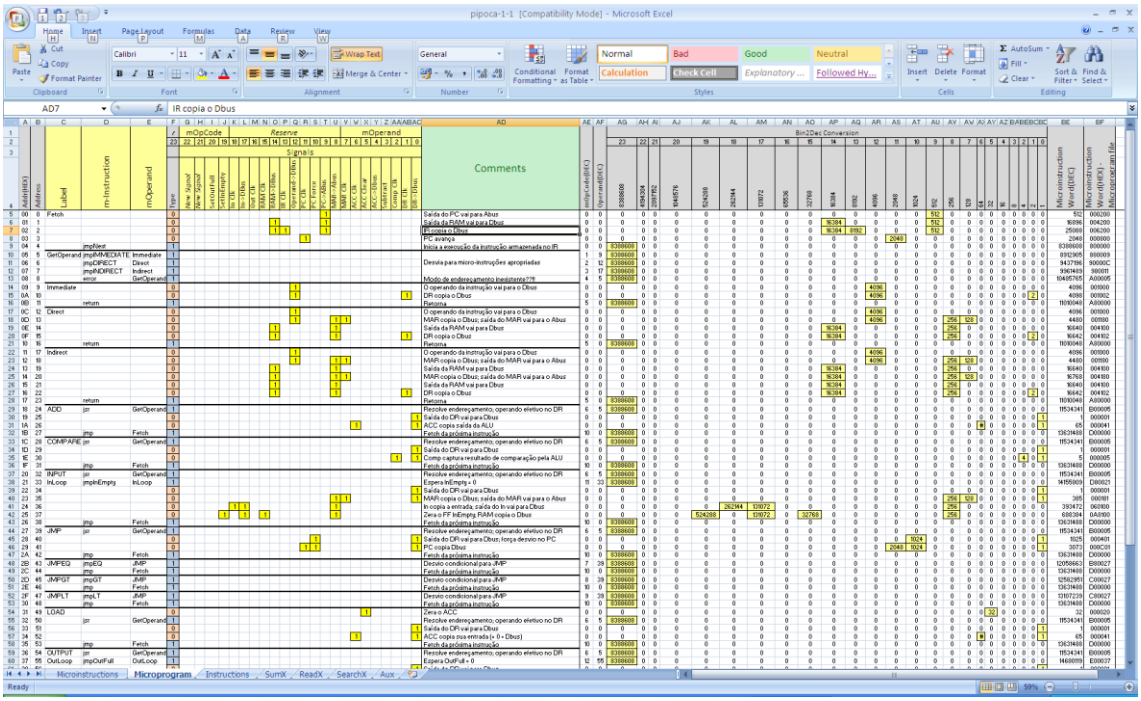


Figura 479: A planilha MicroProgram

O valor em decimal da micro-instrução é igual à soma:

- do campo Type multiplicado por 2^{23}

- do campo OpCode(DEC) multiplicado por 2^{19};
- do campo mOperand.
- das potências de 2 correspondentes a cada sinal igual a 1 na microinstrução.

Deve-se observar que quando Type = 1, a microinstrução é de controle e todos os sinais são iguais a 0, e quando Type = 0 o campo OpCode(DEC) é igual a zero, pois a microinstrução é de sinais. Nas microinstruções de controle os bits de 8 a 18 não estão sendo utilizados.

Coluna Microinstruction Word(HEX). É simplesmente a conversão para hexadecimal com 6 dígitos do valor decimal da microinstrução.

A planilha Instructions está mostrada na Tabela 5, na página 275. Além das colunas Description, Mnemonic, Opcode10 e Opcode2, que já comentamos, esta planilha contém a coluna BranchTable16, que é calculada pesquisando para cada mnemônico de instrução o endereço hexadecimal da micro-instrução com label igual a este mnemônico na planilha Microprogram. É esta coluna que deve ser utilizada para a produção de uma imagem de memória para a ROM Branch Table da Unidade de controle.

1.1.1.1 Programando a Pipoca

As planilhas SumX e ReadX do arquivo Pipoca.xls contêm programas em Assembler, e produzem imagens de memória a serem carregadas na RAM da CPU. Para explicar estas planilhas nós vamos mostrar como se constrói um novo programa.

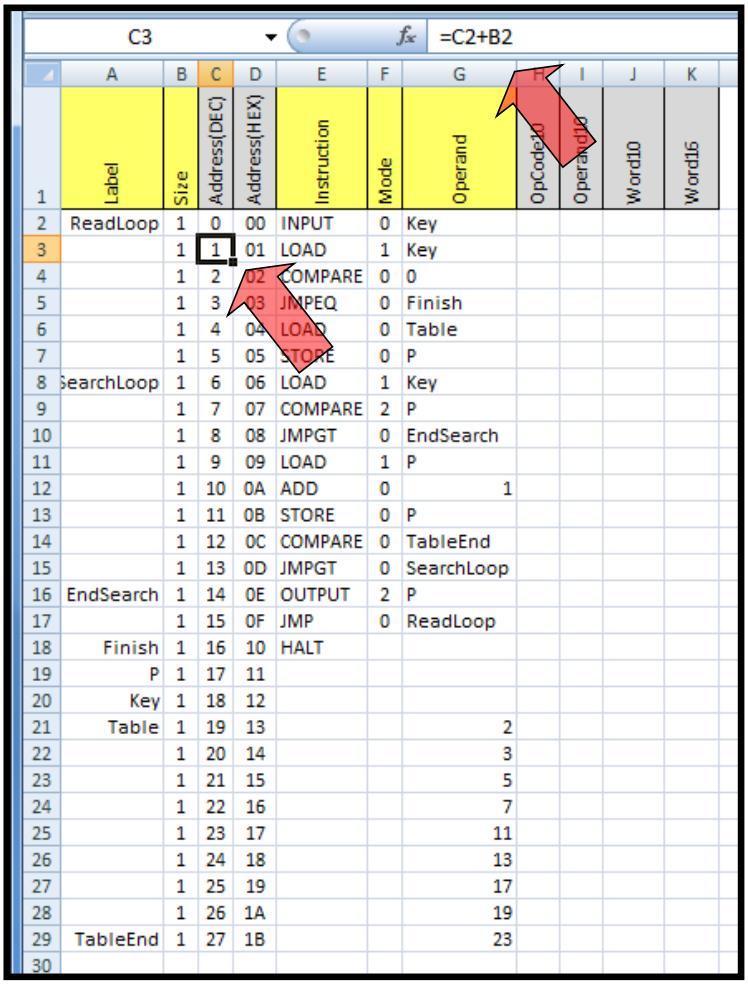
N21

	A	B	C	D	E	F	G	H	I	J	K	L
1	Label	Size	Address(DEC)	Address(HEX)	Instruction	Mode	Operand	OpCode10	Operand10	Word10	Word16	
2	ReadLoop	1			INPUT	0	Key					
3		1			LOAD	1	Key					
4		1			COMPARE	0	0					
5		1			JMPEQ	0	Finish					
6		1			LOAD	0	Table					
7		1			STORE	0	P					
8	SearchLoop	1			LOAD	1	Key					
9		1			COMPARE	2	P					
10		1			JMPGT	0	EndSearch					
11		1			LOAD	1	P					
12		1			ADD	0	1					
13		1			STORE	0	P					
14		1			COMPARE	0	TableEnd					
15		1			JMPGT	0	SearchLoop					
16	EndSearch	1			OUTPUT	2	P					
17		1			JMP	0	ReadLoop					
18	Finish	1			HALT							
19	P	1										
20	Key	1										
21	Table	1					2					
22		1					3					
23		1					5					
24		1					7					
25		1					11					
26		1					13					
27		1					17					
28		1					19					
29	TableEnd	1					23					

Figura 480: O programa SearchX

O primeiro passo consiste em copiar o cabeçalho de um dos programas já existentes em uma nova planilha, e ali codificar o seu programa, como mostrado na Figura 480. Aqui devem ser

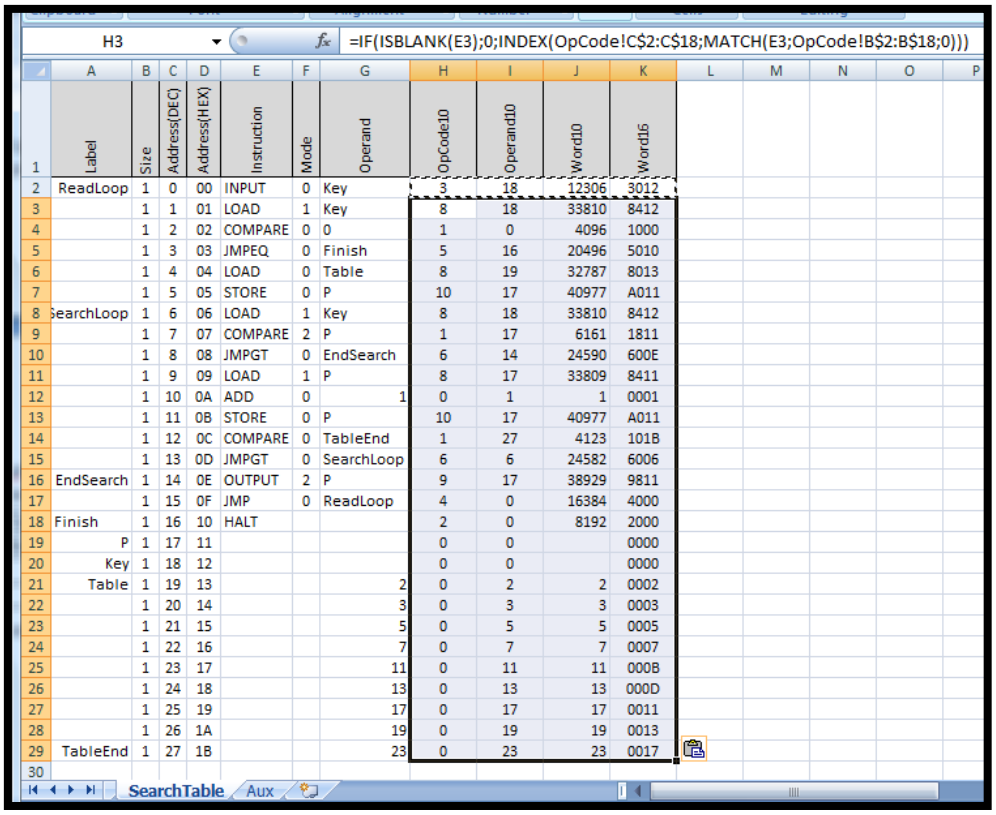
colocados também os tamanhos de cada instrução ou variável – todos iguais a 1 neste programa. Depois, calcule os endereços, colocando 0 (zero) na primeira linha da coluna Address(DEC) e, nas linhas seguintes desta coluna, a fórmula que soma o endereço e o tamanho da linha anterior para obter o endereço da linha corrente, como mostrado na Figura 481. Para obter os endereços em hexadecimal (muito úteis para acompanhar a execução do programa, pois o Logisim mostra endereços em hexadecimal), basta usar a fórmula Excel DEC2HEX(Ci;2) para a linha i da coluna Address(HEX).



	A	B	C	D	E	F	G	H	I	J	K
1	Label	Size	Address(DEC)	Address(HEX)	Instruction	Mode	Operand	OpCode10	Operand10	Word10	Word16
2	ReadLoop	1	0	00	INPUT	0	Key				
3		1	1	01	LOAD	1	Key				
4		1	2	02	COMPARE	0	0				
5		1	3	03	JMPEQ	0	Finish				
6		1	4	04	LOAD	0	Table				
7		1	5	05	STORE	0	P				
8	SearchLoop	1	6	06	LOAD	1	Key				
9		1	7	07	COMPARE	2	P				
10		1	8	08	JMPGT	0	EndSearch				
11		1	9	09	LOAD	1	P				
12		1	10	0A	ADD	0	1				
13		1	11	0B	STORE	0	P				
14		1	12	0C	COMPARE	0	TableEnd				
15		1	13	0D	JMPGT	0	SearchLoop				
16	EndSearch	1	14	0E	OUTPUT	2	P				
17		1	15	0F	JMP	0	ReadLoop				
18	Finish	1	16	10	HALT						
19	P	1	17	11							
20	Key	1	18	12							
21	Table	1	19	13			2				
22		1	20	14			3				
23		1	21	15			5				
24		1	22	16			7				
25		1	23	17			11				
26		1	24	18			13				
27		1	25	19			17				
28		1	26	1A			19				
29	TableEnd	1	27	1B			23				
30											

Figura 481: Cálculo dos endereços - as setas destacam a célula C3 e sua fórmula

Para obter o mapa de memória é preciso agora colocar fórmulas para calcular o valor de cada palavra neste mapa. Para isso, copie as colunas OpCode10, Operand10, Word10 e Word16 da primeira linha do programa SomaX, e cole *somente as fórmulas* nas mesmas colunas da primeira linha do seu novo programa. Depois, selecione as células com as fórmulas no novo programa, copie e cole nas linhas restantes nestas colunas, para obter a planilha completa mostrada na Figura 482.



H3 =IF(ISBLANK(E3);0;INDEX(OpCode!C$2:C$18;MATCH(E3;OpCode!B$2:B$18;0)))

	A	B	C	D	E	F	G	H	I	J	K
1	Label	Size	Address(DEC)	Address(HEX)	Instruction	Mode	Operand	OpCode10	Operand10	Word10	Word16
2	ReadLoop	1	0	00	INPUT	0	Key	3	18	12306	3012
3		1	1	01	LOAD	1	Key	8	18	33810	8412
4		1	2	02	COMPARE	0	0	1	0	4096	1000
5		1	3	03	JMPEQ	0	Finish	5	16	20496	5010
6		1	4	04	LOAD	0	Table	8	19	32787	8013
7		1	5	05	STORE	0	P	10	17	40977	A011
8	SearchLoop	1	6	06	LOAD	1	Key	8	18	33810	8412
9		1	7	07	COMPARE	2	P	1	17	6161	1811
10		1	8	08	JMPGT	0	EndSearch	6	14	24590	600E
11		1	9	09	LOAD	1	P	8	17	33809	8411
12		1	10	0A	ADD	0	1	0	1	1	0001
13		1	11	0B	STORE	0	P	10	17	40977	A011
14		1	12	0C	COMPARE	0	TableEnd	1	27	4123	101B
15		1	13	0D	JMPGT	0	SearchLoop	6	6	24582	6006
16	EndSearch	1	14	0E	OUTPUT	2	P	9	17	38929	9811
17		1	15	0F	JMP	0	ReadLoop	4	0	16384	4000
18	Finish	1	16	10	HALT			2	0	8192	2000
19	P	1	17	11				0	0		0000
20	Key	1	18	12				0	0		0000
21	Table	1	19	13			2	0	2	2	0002
22		1	20	14			3	0	3	3	0003
23		1	21	15			5	0	5	5	0005
24		1	22	16			7	0	7	7	0007
25		1	23	17			11	0	11	11	000B
26		1	24	18			13	0	13	13	000D
27		1	25	19			17	0	17	17	0011
28		1	26	1A			19	0	19	19	0013
29	TableEnd	1	27	1B			23	0	23	23	0017
30											

SearchTable Aux

Figura 482: Planilha completa com o programa SearchTable

As fórmulas para as quatro colunas mais à direita realizam pesquisas e cálculos:

- Para a coluna OpCode10 (OpCode em decimal), a fórmula para a célula H2 é

  H2 =IF(ISBLANK(E2);0;INDEX(Instructions!C$2:C$18;MATCH(E2;Instructions!B$2:B$18;0)))

  que diz ao Excel para colocar 0 se o campo de código de instrução (E2) estiver vazio e, senão, efetuar uma pesquisa na coluna B da planilha Instructions procurando o código da instrução (na coluna E2), obtendo assim seu código decimal, que está na coluna C desta planilha.

- Para a coluna Operand10, a fórmula para a célula I2 é

  I2 =IF(ISBLANK(G2);0;IF(ISNUMBER(G2);G2;INDEX(C$1:C$36;MATCH(G2;A$1:A$36;0);1)))

  dizendo ao Excel para colocar 0 se o campo de operando (G2) estiver vazio e, senão, se o operando for um número, colocar este número diretamente; se o operando for uma referência, pesquisar pelo operando na coluna de labels (A), obtendo o valor do endereço correspondente.

- Para a coluna Word10, a fórmula para a célula J2 é

  J2 = H2*2^12+F2*2^10+I2

  que calcula o valor em decimal da instrução, com potências de 2 escolhidas segundo o posicionamento do termo na palavra de 16 bits como mostrado na **Erro! Fonte de referência não encontrada.**.

- Para a coluna Word16, a fórmula para a célula K2 é

  H2 = DEC2HEX(J2;4)

  que converte o valor decimal para um hexadecimal com 4 dígitos.

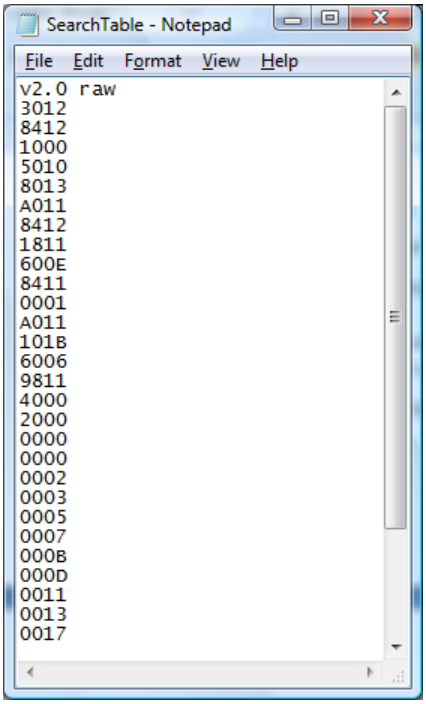


Figura 483: Arquivo com imagem de memória para o programa SearchTable

Para obter um arquivo com o mapa de memória que pode ser lido pelo Logisim,

- use o Bloco de Notas para criar um arquivo SearchTable.txt, e digite “v2.0 raw” em sua primeira linha;
- depois, copie todos os valores na coluna Word16 da planilha com o programa, e cole no Bloco de Notas, a partir da segunda linha. A Figura 483 mostra a janela do Bloco de Notas com o programa executável.

Salve o arquivo; seu programa estará pronto para ser executado.